A Hybrid Static/Dynamic Approach to Scheduling Stream Programs

by

Ceryen Tan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2009

Author .....
Department of Electrical Engineering and Computer Science
August 21, 2009

Certified by .....
Saman Amarasinghe
Professor
Thesis Supervisor

Accepted by .....
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

A Hybrid Static/Dynamic Approach to Scheduling Stream Programs

by

Ceryen Tan

Submitted to the Department of Electrical Engineering and Computer Science
on August 21, 2009, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

Streaming languages such as StreamIt are often utilized to write stream programs that execute on multicore processors. Stream programs consist of actors that operate on streams of data. To execute on multiple cores, actors are scheduled for parallel execution while satisyfing data dependencies between actors. In StreamIt, the compiler analyzes data dependencies between actors at compile-time and generates a static schedule that determines where and when actors are executed on the available cores.

Statically scheduling actors onto cores results in no scheduling overhead at run-time and allows for sophisticated compile-time scheduling optimizations. Unfortunately, static scheduling has a number of severe limitations. The generated static schedule is inflexible and cannot be adapted to run-time conditions, such as cores that are unexpectedly unavailable. Static scheduling may also incorrectly load-balance cores due to inaccurate static work estimates.

This thesis contributes a hybrid static/dynamic scheduling approach that attempts to address the limitations of static scheduling. Dynamic load-balancing is utilized to adjust the static schedule to run-time conditions and to correct load imbalances that might exist after static scheduling. Dynamic load-balancing is designed to add very little run-time overhead.

Thesis Supervisor: Saman Amarasinghe
Title: Professor

Acknowledgments

I am personally indebted to Michael Gordon for his contributions to this thesis. Much of the SMP compiler backend described in this thesis was adapted from Michael's work with the Tilera multicore architecture. Michael was also an invaluable mentor, providing sound advice and guidance all throughout this thesis. This thesis would not have been possible without his assistance.

I would like to thank William Thies for his remote assistance and his vast technical knowledge. I would also like to thank my advisor, Saman Amarasinghe, for directing and supporting my work.

Finally, I would like to thank my loving and supportive friends and family. I dedicate this thesis to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The emergence of multicore architectures has dramatically changed the computing landscape. No longer can programmers depend on hardware-based improvements to single-threaded performance, the key to improved performance now lies in parallelization. With multicore architectures becoming increasingly prevalant in every corner of the computing market, exploiting parallelism in applications has become imperative if programmers want to take full advantage of current and future architectures.

Unfortunately, developing parallel applications is still very much a daunting task. Traditional languages such as C were designed for the serial execution of instructions and contain no built-in facilities to express parallelism. As a result, compilers are largely unable to assist the programmer in parallelizing an application as analyzing serial code for parallelism requires herculean effort. The task of parallelization therefore lies primarily in the hands of the programmer. Parallelization requires the programmer to contend with complicated issues such as races, synchronization and communication overhead, and low-level architectural details. The sheer difficulty of writing effective parallel code means that only a select group of professionals in the computing industry actively engage in developing parallel applications.

In this context, streaming languages are an attractive alternative to traditional imperative languages [5, 16, 21]. In streaming languages, the programmer defines actors that operate on streams of data. The programmer then connects actors via explicit communication channels to form stream graphs. Because programs are ex-

plicitly structured as communicating actors, programs typically expose a high degree of parallelism. This exposed parallelism can be exploited by the compiler to automatically parallelize programs on behalf of the programmer, which frees the programmer from spending significant time and effort on issues such as low-level implementation, performance tuning, and non-deterministic bugs.

StreamIt [27] is a streaming language and compilation infrastructure developed at MIT. The StreamIt language is characterized by powerful features that help to expose structure and parallelism within stream programs, while encouraging code reuse and improving programmer productivity. The StreamIt language aims to be machine-independent, while the StreamIt compiler aims to generate high-performance parallel programs for a wide range of multicore architectures.

Parallelization of a stream program is achieved by scheduling actors to execute in parallel while satisfying data dependencies between actors. Ideally, with enough parallelism in a stream program, actors can be scheduled such that cores are load-balanced and utilized at all times. Numerous scheduling algorithms have been developed, ranging from fully static scheduling to fully dynamic scheduling [14]. Fully static scheduling analyzes data dependencies between actors at compile-time and generates a static schedule that determines where and when each actor should be executed on the available cores. Fully dynamic scheduling tracks data dependencies between actors at run-time, executing actors in parallel when their data dependencies are satisfied.

StreamIt makes use of fully static scheduling to parallelize stream programs[1]. Fully static scheduling has a number of significant advantages. Because compilation is not performance-critical, the compiler has a considerable amount of time to optimize the generated static schedule. For example, the compiler can afford to spend time globally minimizing the amount of communication necessary to execute the static schedule. Because the static schedule is generated at compile-time, fully static scheduling has no run-time cost; processing time is spent entirely on computation

[1]While there is partial support for dynamic I/O rates in StreamIt [7], this support nonetheless relies on a static mapping of filters to processors.

and not on scheduling. The static schedule can also be executed in a decentralized manner as each core knows a priori which actors it will execute.

Unfortunately, fully static scheduling comes with a number of significant disadvantages. Fully static scheduling is inflexible and cannot adapt to run-time conditions. If fewer cores are available than expected or cores are shared with other processes, the static schedule attempts to utilize processing resources that are unavailable, causing overall performance to suffer. In addition, fully static scheduling depends on accurate static work estimates for actors to statically load-balance the cores. Unfortunately, static work estimates for actors are often inaccurate, which impacts the effectiveness of static load-balancing. Even if cores are perfectly load-balanced according to static work estimates, cores may be severely imbalanced at run-time.

Fully dynamic scheduling overcomes these problems by making scheduling decisions at run-time. Cores that are ready for work request actors to execute from the dynamic scheduler. Cores that are unavailable or shared with other processes are handled automatically as they simply request fewer or no actors to execute. Fully dynamic scheduling also keeps cores automatically load-balanced as cores continually execute actors if actors are available to execute. However, fully dynamic scheduling has its own disadvantages. Tracking data dependencies between actors is a significant overhead that can impair run-time performance. In addition, fully dynamic scheduling cannot perform sophisticated scheduling optimizations such as globally minimizing communication. These scheduling optimizations would be too time-consuming to perform at run-time. Fast and simple heuristics must be used instead to mimic the effect of these scheduling optimizations.

This thesis proposes a new scheduling approach called *static scheduling with dynamic adjustments*. This scheduling approach is a hybrid between static scheduling and dynamic scheduling. Static scheduling is first utilized to generate an optimized, mostly load-balanced static schedule. This static schedule is then periodically adjusted at run-time using dynamic load-balancing. Dynamic load-balancing adjusts the static schedule by moving work between cores to keep cores load-balanced. If cores become unavailable or are shared with other processes, dynamic load-balancing

adjusts the static schedule to compensate by moving work away from these cores. Dynamic load-balancing also adjusts the static schedule to correct any load imbalances that might exist after static load-balancing.

This scheduling approach is unique in that most of the scheduling computation is completed at compile-time with inexpensive dynamic adjustments at run-time. Dynamic adjustments are inexpensive as only a relatively small number of schedule modifications are sanctioned by the compile-time analysis. Because dynamic adjustments are inexpensive, there is very little run-time overhead, in contrast to fully dynamic scheduling. In addition, because most of the scheduling computation is completed at compile-time, sophisticated compile-time optimizations can be applied. This scheduling approach therefore leverages the advantages of both static scheduling and dynamic scheduling, while avoiding many of the disadvantages.

This scheduling approach was developed in the context of StreamIt. In particular, it was implemented in a new StreamIt compiler backend that targets SMP machine. This new SMP compiler backend was developed in conjunction with this thesis. SMP machines often run numerous processes that compete for processing time, making fully static scheduling ineffective. The implementation of static scheduling with dynamic adjustment within the new SMP backend is therefore very appropriate.

Because this scheduling approach allows for compile-time optimizations, optimizations were developed during the course of this thesis that reduce the communication overhead for *filter fission*, an important technique used to split the work of data-parallel filters across multiple cores. These optimizations on filter fission are detailed in this thesis.

This thesis specfically contributes the following items:

- A new StreamIt compiler backend that specifically targets SMP machines.
- A run-time infrastructure that allows a static schedule to be adjusted at run-time, allowing work to be moved between cores.
- A dynamic load-balancing algorithm that adjusts the static schedule to respond to run-time conditions. This dynamic load-balancing algorithm attempts to cor-

rect load imbalances that exist in the static schedule after static load-balancing. This dynamic load-balancing algorithm also attempts to compensate for cores that are unavailable or shared with other processes.

- Optimizations that reduce the communication overhead of filter fission

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of the StreamIt language, while Chapter 3 describes the new SMP compiler backend in great detail. Chapter 4 describes the infrastructure developed to allow the static schedule to be adjusted at run-time. Chapter 5 describes the dynamic load-balancing algorithm used to adjust the static schedule at run-time. Chapter 6 describes the optimizations on filter fission. Chapter 7 evaluates the performance of static scheduling with dynamic adjustments, as well as the performance of the optimized filter fission.

Chapter 2

The StreamIt Language

Streaming languages are specialized languages designed for stream processing. Streaming languages are based around the concept of actors. Actors are processing blocks that transform input data streams into output data streams. Actors can be connected together via explicit communication channels to form complex computation networks. The composition of actors into computation networks allows for sophisticated stream processing [6].

This model of computation is appropriate for a wide range of applications. For example, signal processing applications generally exhibit block-diagram structure where blocks perform processing on data streams. Such applications are easily conceptualized in terms of stream-processing actors, making streaming languages a natural choice. Streaming languages have been used to write many high-performance applications in audio, video, and image processing.

StreamIt [27] is a high-performance streaming language and compilation infrastructure developed at MIT. StreamIt is heavily based on the synchronous dataflow (SDF) model of computation, where actors are constrained to known, fixed communication rates. This restriction makes it easier for the compiler to analyze stream graphs, allowing for efficient scheduling of actors for parallel execution [20]. StreamIt extends SDF by imposing hierarchical structure on stream graphs. This hierarchical structure helps to expose parallelism within stream graphs, while encouraging code reuse and improving programmer productivity. StreamIt has been used to express

a number of complex applications, such as MPEG2 encoding and decoding [8] and image-based motion estimation [2].

2.1 Filters

In StreamIt, the basic unit of computation is a filter. Filters are actors with a single input tape, a single output tape, and a user-defined work function for translating input items to output items. A filter executes by reading data elements from the input tape, processing via the work function, and writing results to the output tape. Input and output tapes are FIFO queues that connect filters. The work function reads data from the input tape using the operations of `pop()` and `peek(index)`, where `peek` returns the data element at position `index` in the input tape without dequeuing the element. The work function writes data to the output tape using the operation of `push(value)`. The number of elements pushed, popped, and peeked by a single invocation of the work function must be declared. In this work, we consider the case where these *push*, *pop* and *peek* rates are constrained to be static and fully known at compile time.

Filters may contain fields and arbitrary helper functions. A special function named `init` can be used to perform one-time initialization of constants. Another special function named `prework` can be used to perform initial work. This function is called exactly once at the start of program execution, before the `work` function is ever executed. The `prework` function can access the input and output tapes and can have different I/O rates from the `work` function. Note that the user never calls the `work`, `init`, and `prework` functions directly as they are called automatically. Figure 2.1 illustrates a filter with `work`, `init`, and `prework` functions. Note that the `work` and `prework` functions declare separate *push*, *pop*, and *peek* rates. The filter also contains a single field initialized by *init* and a helper function.

Filters that contain fields modified by the `work` function are known as stateful filters. These are filters that maintain state outside of the data elements buffered in the tapes. This state introduces internal dependencies between one execution of a

```
float->float filter FIRFilter(float sampleRate, int N) {
    float[N] weights;

    init {
        weights = calculateImpulseResponse(sampleRate, N);
    }

    prework push N-1 pop 0 peek N {
        for (int i=1 ; i<N ; i++) {
            push(doFIR(i));
        }
    }

    work push 1 pop 1 peek N {
        push(doFIR(N));
        pop();
    }

    float doFIR(int k) {
        float val = 0;
        for(int i=0 ; i<k ; i++) {
            val += weights[i] * peek(k-i-1);
        }
        return val;
    }
}
```

Figure 2-1: Example StreamIt filter

stateful filter and the next. In contrast, filters that do not contain fields modified by the `work` function are known as stateless filters. Stateless filters have no internal dependencies between executions. Because they have no internal dependencies between executions, stateless filters are an important source of parallelism within a stream program. This will be discussed later throughout the remainder of this thesis.

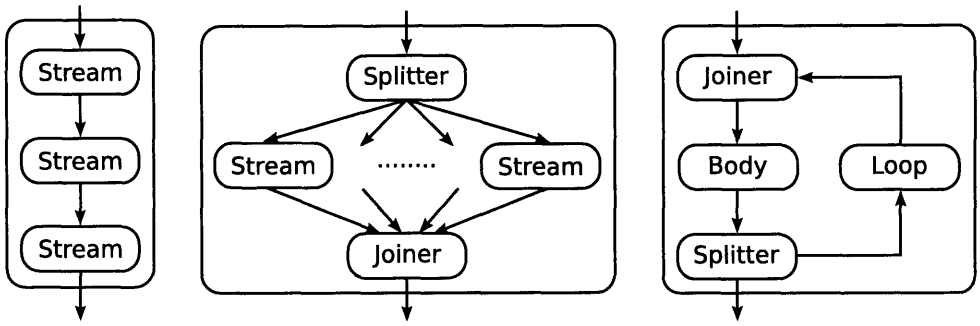
Note that many stateful filters can be rewritten as stateless filters through the use of peeking. Filters often perform sliding window computations on input data (e.g. FIR filters). A naive filter implementation of a sliding window computation would store the current window of data in the filter, which would introduce state into the filter. A better implementation would simply use the first *peek* elements on the input tape to represent the current window of data to process. This implementation eliminates the need for state, allowing what would normally be a stateful filter to be stateless. Because the first *peek* elements of an input tape are often used as a sliding data window, we refer to these elements as the *peeking window* for a filter.

2.2 Hierarchical Composition of Filters

StreamIt provides three constructs for composing filters: *pipeline*, *splitjoin*, and *feedbackloop*. Much like filters, each construct has a single input tape and a single output tape. This allows for arbitrarily deep hierarchical composition where each construct can be a composition of both filters and other constructs. To simplify further discussion, we define a *stream* to be any instance of a filter, pipeline, splitjoin, or feedbackloop.

The *pipeline* is the most basic StreamIt construct. A pipeline is used to connect streams in sequence, with the input tape of a stream connected to the output tape of the previous stream in the pipeline.

The *splitjoin* construct is used to specify independent parallel streams with a common *splitter* and a common *joiner*. A *splitter* splits an input tape into multiple output tapes. StreamIt presently allows for two types of splitters: 1) *duplicate*, which duplicates each element from the input tape to each output tape, and 2)



(a) Pipeline (b) Splitjoin (c) Feedbackloop

Figure 2-2: StreamIt hierarchical structures

roundrobin$(w_0, ..., w_n)$, which distributes elements in a weighted round-robin fashion. The output tapes from a splitter are connected to the input tapes of the parallel streams. The outputs of these parallel streams are then fed into a *joiner*, which combines the output tapes of each parallel stream into a single output tape. StreamIt joiners are presently restricted to join elements in a weighted round-robin fashion.

The *feedbackloop* construct provides a way to create cycles in a stream graph. The feedbackloop construct consists of a forward path and a feedback path. Along the forward path is a *body* stream which performs computation on elements passing through the forward path. Along the feedback path is a *loop* stream which performs computation on elements passing through the feedback path. The feedbackloop construct also contains a splitter, which distributes elements from the forward path between the output tape and the feedback path, and a joiner, which merges elements from the feedback path with elements from the input tape.

Unfortunately, newer techniques in the StreamIt compiler presently lack support for the feedbackloop construct. This thesis therefore focuses on stream programs built entirely from pipelines and splitjoins. Figure 2-3 illustrates an FM radio application that is built using these constructs. This application takes an analog signal from an antenna and demodulates a single FM channel. This is followed by equalization of the FM channel.

```
void->void pipeline FMRadio(int N, float lo, float hi) {
  add AtoD();
  add FMDemod();
  add splitjoin {
    split duplicate;
    for (int i=0; i<N; i++) {
      add pipeline {
        add LowPassFilter(lo + i*(hi - lo)/N);
        add HighPassFilter(lo + i*(hi - lo)/N);
      }
    }
    join roundrobin;
  }
  add Adder();
  add Speaker();
}
```

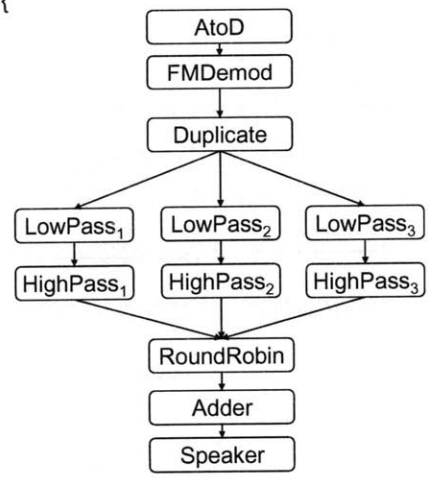


Figure 2-3: FM radio application built from pipelines and splitjoins

2.3 Execution Model

A StreamIt program consists of a top-level stream that defines a hierarchical stream graph. At the leaves of the hierarchy are filters, connected together by channels. The top-level stream is implicity surrounded by an infinite loop and the stream graph is executed until all data is exhausted from data sources. Data sources (i.e. files) feed data into the stream graph, while data sinks drain data from the stream graph.

Filter executions are interleaved according to data dependencies. If a filter has sufficient input data in its input channel and room in its output channel, the filter can execute its work function. Execution of a filter's work function must run to completion. The work function may be executed multiple times if the filter has enough input data and room in its output channel. After the completion of a filter's work function, execution may switch to another filter.

2.4 Exposed Parallelism

The StreamIt language helps to expose three types of parallelism within a stream program:

- Task parallelism: The child streams of a splitjoin are task parallel as the output of one child stream never reaches the input of another. Task parallelism reflects the logical parallelism in the underlying algorithm.

- Data parallelism: Stateless filters are data parallel. Because stateless filters maintain no state, a stateless filter has no dependencies between one execution and the next. This allows multiple instances of the same stateless filter to be executed simultanously on different parts of an input data stream.

- Pipeline parallelism: The child streams of a pipeline are pipeline parallel.

2.5 StreamIt Extensions

The full StreamIt language provides a number of important features beyond the basic model described here. These features greatly extend the power of the StreamIt language, allowing for more complex applications. In particular, the StreamIt language allows for dynamic I/O rates for filters and teleport messaging for out-of-band communication between filters [28]. For the purposes of this thesis, these features are presently unsupported.

Chapter 3

The StreamIt Compiler

The StreamIt compiler is a source-to-source compiler that takes StreamIt code and generates C/C++ code. The generated code contains a parallel implementation of the input StreamIt code. The goal of the StreamIt compiler is to maximize the parallel performance of the generated code. Parallelization is achieved through the static scheduling of filters to available cores. Static scheduling exploits all three forms of parallelism in the stream graph – task, data, and pipeline parallelism – to maximize parallel performance. Static scheduling load-balances the cores through the use of static work estimates. Static scheduling also attempts to globally minimize the amount of communication necessary between cores.

The StreamIt compiler contains a number of backends that each targets a specific architecture. Each backend generates parallel code specific the architecture that it targets. Existing backends presently target MIT Raw [10], Tilera TILE64, IBM Cell [31], GPUs [29, 30], and computer clusters. This chapter details a new SMP compiler backend that generates parallel code for commodity multicore architectures. These architectures are generally characterized by a shared memory model, multi-level cache hierarchy, and instruction-level streaming enhancements. These features make commodity multicore architectures significantly different from many of StreamIt's previous targets.

The parallel code generated by the SMP compiler backend attempts to take advantage of these special architectural features. Shared memory is utilized to communicate

Phase	Function
KOPI Front-end	Parses syntax into a Java-like abstract syntax tree.
SIR Conversion	Converts the AST to the StreamIt IR (SIR).
Graph Expansion	Expands all parameterized structures in the stream graph.
Filter Scheduling	Calculates initialization, prologue, and steady-state filter execution orderings.
Partitioning	Performs fission and fusion transformations for load-balancing.
Layout	Determines minimum-cost placement of filters on cores.
Communication Scheduling	Orchestrates fine-grained communication between cores.
Code generation	Generates code for cores.

Table 3.1: Phases in the StreamIt compiler

data between cores. Shared caches between neighboring cores are utilized whenever possible to minimize the cost of communicating data through shared memory. The generated code is structured to be easily vectorizable, which allows for the utilization of instruction-level streaming enhancements.

3.1 Compiler Phases

Compilation of a StreamIt program is broken down into multiple phases. These phases are outlined in Table 3.1. The first three phases are responsible for parsing a StreamIt program into an internal representation called StreamIt IR (SIR) that encapsulates the hierarchical stream graph of the program. Parameterized stream constructs are expanded into static structures, yielding a fully static hierarchical stream graph in the SIR. The first three phases are shared amongst all StreamIt compiler backends and are not detailed further in this thesis.

The remaining phases are unique to each StreamIt compiler backend. Filter scheduling determines an ordering on filter executions that will satisfy data dependencies between filters. Partitioning divides the stream graph into load-balanced partitions that can be assigned to cores, utilizing fission and fusion transformations on the stream graph to perform load-balancing across partitions. Layout assigns partitions to cores, attempting to globally minimize communication costs. Communication scheduling orchestrates the fine-grained movement of data between cores. Finally, code generation generates parallel code for the target architecture. The following sections describe these phases in the context of the new SMP compiler backend.

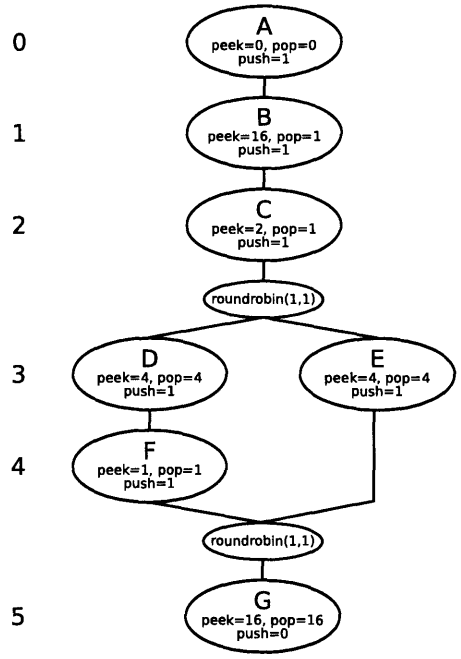


Figure 3-1: Levels of a simple stream graph

3.2 Stream Graph Levels

Various passes in the compiler operate on the notion of stream graph levels. The first level l_0 of a stream graph consists of source filters. Each successive level l_i is the set of filters whose input dependencies are satisfied by previous levels l_j where $j < i$. Figure 3-1 illustrates stream graph levels on a simple stream graph. Dividing the stream graph into levels is useful as each level contains filters that are task parallel. Levels also simplify the analysis of input dependencies since each level depends only on the outputs of previous levels.

Take a filter that is in level l_j and another filter that is in level l_k. We define the first filter to be *downstream* of the second filter if $j > k$. Conversely, we define the first filter to be *upstream* of the second filter if $k > j$. In Figure 3-1, filter G is downstream of filter A.

3.3 Filter Scheduling

Filter scheduling is responsible for determining an ordering of filter executions that will satisfy data dependencies between filters. Filter scheduling generates three schedules for filter execution: initialization, prologue, and steady-state. This section details the purpose and construction of these schedules.

3.3.1 Steady-State and Initialization

Because a StreamIt program is surrounded by an implicit infinite loop, the StreamIt compiler needs to generate a periodic schedule of filter executions that can be looped indefinitely. This periodic schedule must have the property that the number of data elements buffered on each channel is unchanged after the execution of each period. Otherwise, continued execution of the periodic schedule would cause the number of elements on each channel to either grow without bound or shrink to zero during execution. The former is undesirable as we would like to keep buffering bounded on each channel. The latter is undesirable as it would prevent further execution of the periodic schedule.

A single period of the periodic schedule is known as a *steady-state schedule.* A steady-state schedule is a sequence of filter executions with the above property of leaving the number of data elements buffered on each channel unchanged. A number of algorithms have been designed to construct the steady-state schedule. These algorithms trade between schedule complexity, buffering requirements, and latency by generating different orderings of filter executions [17, 23, 3, 12].

The SMP compiler backend uses a Single Appearance Schedule [10] for the steady-state. This schedule makes no effort to minimize buffering requirements, which isn't a major concern for SMP machines. The advantage of this schedule is that it is very simple. Filters are ordered such that upstream filters execute before downstream filters, which is necessary to ensure that downstream filters have input to process. Each filter appears exactly once in the schedule and each filter is assigned a *multiplicity*, the number of iterations for the filter in the schedule.

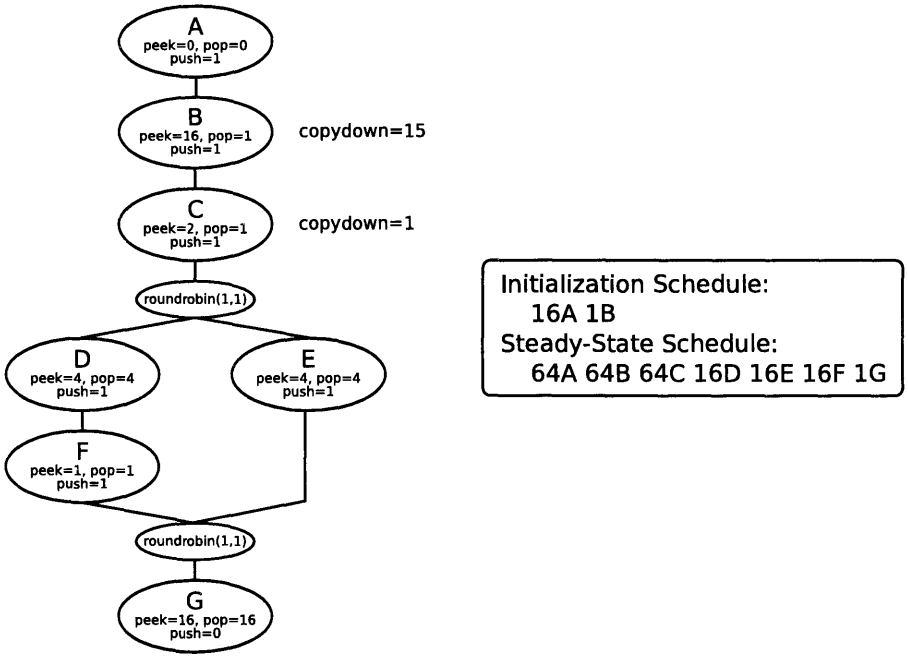


Figure 3-2: Initialization and Steady-state schedules for a simple stream graph

Some initialization may be necessary before the steady-state schedule can be executed. Filters that have higher *peek* rates than *pop* rates are known as *peeking filters.* These filters require at least *peek* - *pop* elements on their input channels before the steady-state schedule can be executed. This is so that after upstream filters have executed in the steady-state schedule, peeking filters have at least *peek* elements on their input channels to satisfy their peeking windows. Placing at least *peek* - *pop* elements on the input channels of peeking filters is performed by an *initialization schedule.* The initialization schedule is constructed by symbolically executing the stream graph until each filter has at least *peek* - *pop* input elements. The exact number of elements left on each input channel, which must be at least *peek* - *pop*, is known as *copydown.*

Figure 3-2 illustrates the initialization and steady-state schedules of a simple stream graph. Because filters B and C are peeking filters, the initialization schedule is constructed such that at least *peek* - *pop* elements are left on their respective input buffers after initialization. These elements are necessary for the peeking windows of B and C. As previously described, the number of elements left on each input buffer af-

ter initialization is termed *copydown*. Also, as previously described, the steady-state schedule is constructed such that the number of elements buffered on each channel does not change.

Because the steady-state schedule is meant to be repeated indefinitely, we define a single execution of the steady-state schedule to be a *steady-state iteration*. Note that the steady-state schedule only dictates the ordering of filter executions and the number of iterations for each filter within a single steady-state iteration. The steady-state schedule does not dicate the cores on which filters must be executed. This is handled later by partitioning and layout.

3.3.2 Software Pipelining

As previously described, filters are ordered in the steady-state schedule such that downstream filters execute after upstream filters. This is necessary as downstream filters must wait for output from upstream filters before they can execute.

Software pipelining is a technique explored in [11] that removes the need for filter ordering within a steady-state schedule. Before execution of the steady-state schedule, software pipelining increases the number of elements on each input channel such that each filter can execute its full set of iterations without first waiting on upstream filters. This removes all ordering dependencies within a steady-state schedule, reducing the steady-state schedule to simply the number of iterations for each filter. Within a single steady-state execution, each filter can execute at any time relative to other filters, and may even execute in parallel with other filters. Note that filters are still ordered between steady-state executions as all filter iterations in one steady-state execution must be completed before the next steady-state execution can begin.

Software pipelining adds tremendous flexibility to scheduling. Without software pipelining, scheduling must ensure execution ordering by either assigning neighboring filters to the same core or inserting synchronization between filter executions. With software pipelining, execution orderingis not an issue and scheduling can simply assign filters to cores arbitrarily. This allows for better load balancing as filters can be more easily distributed across cores.

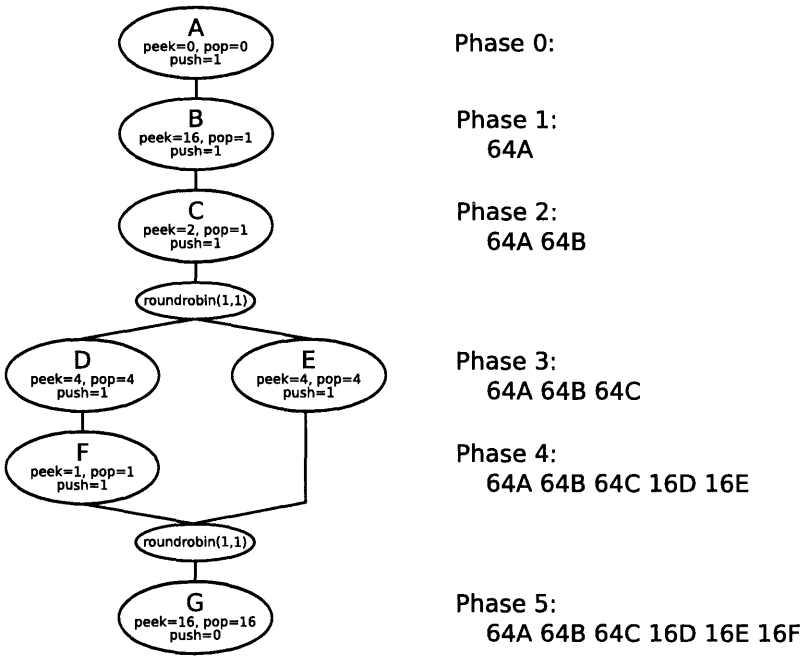


Figure 3-3: Prologue schedule for a simple stream graph. Each phase of the prologue schedule increments the steady-state iteration for levels above the phase. After all phases are executed, each level is a steady-state iteration ahead of the successive level.

Unfortunately, software pipelining comes with some potential pitfalls to be aware of. Software pipelining can dramatically increase the number of elements that are buffered on each input channel. If there are too many elements buffered on an input channel, some negative cache effects may be experienced. Ideally, the buffer for an input channel would be small enough to stay resident in cache. If the buffer for an input channel is made too large to fit into cache, this makes it more difficult for cache to exploit temporal locality.

Software pipelining generates a *prologue schedule* that is executed after initialization and before steady-state. The prologue schedule advances the steady-state iteration of each stream graph level such that each level is an iteration ahead of the successive level. Advancing each level an iteration ahead of the successive level ensures that the successive level has enough input to execute at any given time. Subsequent execution of the steady-state schedule maintains this separation between levels.

The prologue schedule is broken into multiple phases, one phase for each level of

the stream graph. Phase i advances the steady-state iteration of levels above level i. After execution of all phases, each level is an iteration ahead of the successive level. The phases of a prologue schedule is illustrated in Figure 3-3.

3.4 Partitioning

Partitioning divides the stream graph into n roughly load-balanced partitions, where n is the target number of cores. These partitions are later assigned to cores by layout in section 3.5. Partitions are load-balanced through the use of filter fission and filter fusion. This section first describes the techniques of filter fission and filter fusion. This section then describes how these techniques are used to divide the stream graph into n load-balanced partitions.

3.4.1 Filter Fission

Stateless filters are an important source of data parallelism within streaming applications. Because stateless filters have no dependencies from one iteration to the next, multiple instances of the same stateless filter can be executed simultaneously on different parts of an input data stream. Streaming applications are composed mostly of stateless filters [26]. This makes data parallelism widely available within most programs.

Partitioning exploits the data parallelism of stateless filters using a process called *filter fission*. Filter fission clones a stateless filter into multiple partitions. Each partition then executes the stateless filter on a subset of the input data stream. This process divides the work of a stateless filter across multiple partitions, allowing the stateless filter's work to be executed in parallel.

Previous methods for filter fission involved a stream graph transformation that replaced a stateless filter with a splitjoin of clones [10]. The steady-state work for the filter was divided evenly across these clones by evenly distributing the filter's steady-state multiplicity. Each clone could then be assigned to a different partition. The splitjoin was used to ensure that each clone executes on a different subset of the

input data stream. This technique was suitable for distributed-memory machines.

Filter fission in the SMP backend involves no actual transformation on the stream graph. The steady-state multiplicity of a stateless filter is simply divided evenly across multiple partitions, giving each partition an equal number of the stateless filter's iterations. Code generation is then responsible for ensuring that each partition executes on a different part of the input data stream, as will be discussed in section 3.7.3.

3.4.2 Filter Fusion

Fusion is a filter transformation first explored in [10]. Fusion can combine neighboring filters into a single filter. Fusion can also collapse a splitjoin into a single filter. Fusion enables powerful inter-filter optimizations such as scalar replacement [25] and algebraic simplification [1, 18].

Fusion is utilized in two ways during partitioning. Consider a pipeline of stateless filters. Each stateless filter could be individually fissed. Unfortunately, fission introduces communication overhead as data must be distributed to and from separate cores. Fissing each individual filter would therefore introduce excessive communication overhead. Fusion is used to address this by first fusing the pipeline of stateless filters into a single stateless filter. This single stateless filter can then be fissed with significantly less communication overhead than fissing each individual filter. This optimization was first explored in [11] and is applied to all pipelines of stateless filters in a stream graph.

Note that the fusion of two stateless filters sometimes results in a stateful filter, which cannot be subsequently fissed. This is undesirable as it reduces the amount of data parallelism available in a program. Pipelines of stateless filters are therefore fused as much as possible as long as the result of fusion is stateless. Also note that a pipeline may contain stateless splitjoins. These stateless splitjoins can be fused along with other stateless filters in the pipeline.

After filters are assigned to partitions, fusion is applied to neighboring stateful filters that are in the same partitions. This helps to reduce the amount of generated

communication code, while enabling the powerful inter-filter optimizations previously described. Note that this is only applied to stateful filters as stateless filters may be fissed across multiple partitions.

3.4.3 Partitioning Stages

Partitioning in the SMP backend is composed of four stages. In the first stage, pipelines of stateless filters are fused as described in section 3.4.2. In the second stage, the partitioner looks at stateless filters in the stream graph. Each stateless filter is simply fissed across all n partitions. Each partition inherits an equal number of iterations for the stateless filter, keeping the partitions load-balanced.

The partitioner then looks at stateful filters in the stream graph. Stateful filters cannot be fissed and can only be executed on one core at a time. Stateful filters are bin-packed across the n partitions, which involves sorting the stateful filters by static work estimations, then iteratively assigning the largest unassigned filter to the partition with the least amount of work. Bin-packing roughly load-balances the stateful filters across the partitions. Note that bin-packing is made possible by software pipelining as software pipelining allows any pair of filters to be executed in parallel. Finally, neighboring stateful filters in the same partition are fused together as described in section 3.4.2.

Figure 3-4 illustrates partitioning for a simple stream graph. In this figure, filters A, G and C are stateful filters, while all other filters are stateless. Stateless filters are fissed across all partitions, which evenly divides their work, while stateful filters are bin-packed. Note that while partitioning attempts to keep partitions load-balanced, the load-balancing isn't perfect. In particular, the stateful filters can't be perfectly load-balanced, which adds a slight imbalance to the partitions. A smarter partitioning algorithm would fix the imbalance by dividing stateless filter iterations unequally across partitions. However, this depends heavily on accurate static work estimates. If static work estimates are inaccurate, then dividing stateless filter iterations unequally across partitions may actually add to the load imbalance rather than reducing it.

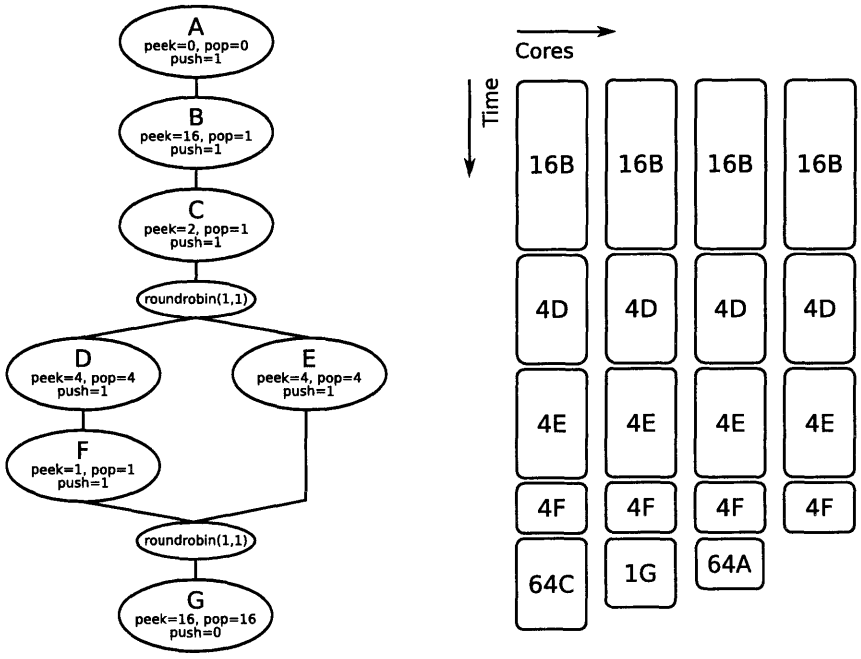


Figure 3-4: Partitioning for a simple stream graph.

3.5 Layout

The end result of partitioning is n roughly load-balanced partitions. Layout assigns partitions to cores such that the cost of communication between partitions is minimized. Communication in the SMP backend makes use of shared memory. Neighboring cores with shared cache have relatively inexpensive communication since data passes through the shared cache rather than through main memory and the shared data bus. Layout therefore attempts to place partitions that communicate most with each other on neighboring cores. This takes advantage of shared caches if they are available. The end result of layout is a core assignment for each partition, which in turn assigns filters to each core.

3.6 Communication Scheduling

At this stage, the compiler has finished assigning filters to cores. However, static scheduling is not yet complete. Static scheduling also schedules the movement of

data between filters.

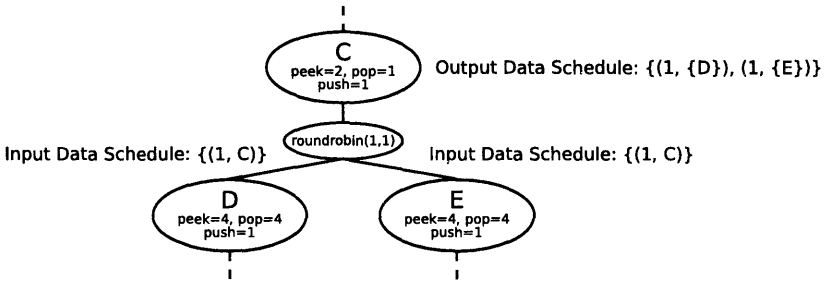
Splitters and joiners act as bottlenecks during execution as they are centralized points through which data must pass. To eliminate the need for splitters and joiners, input and output data schedules are calculated for each filter. The input data schedule determines the exact filter source for each input element. The output data schedule determines the exact filter destinations for each output element. Calculating these data schedules allows data to be transmitted directly between filters without relying on centralized splitters and joiners.

An input data schedule is a round-robin schedule that consists of weights $(w_0, ..., w_n)$ and a source filter assigned to each weight $(f_0, ..., f_n)$. This schedule indicates that the first w_0 input elements are received from filter f_0, the next w_1 input elements are received from filter f_1, etc.

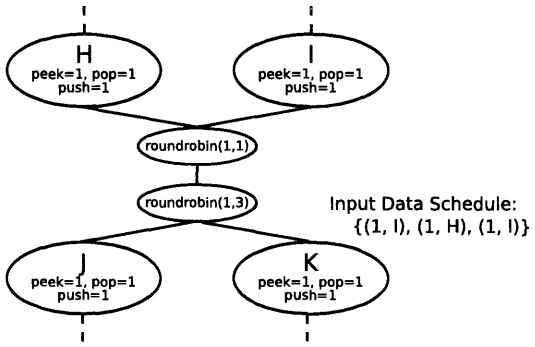
An output data schedule is a round-robin schedule that consists of weights $(w_0, ..., w_n)$ and a destination filter set assigned to each weight $(s_0, ...s_n)$. This schedule indicates that the first w_0 output elements are sent to filter set s_0, the next w_1 output elements are sent to filter set s_1, etc.

The algorithms for extracting these data schedules from the stream graph are beyond the scope of this thesis, though they have been fully implemented in the StreamIt compiler. For many filters, it is trivial to determine the input and output data schedules. For example, in Figure 3-5a, it is clear that filters D and E receive all input from filter C. It is also clear that the destination for filter C's output alternates between filters D and E. The input and output data schedules in this case are very simple. Note that input and output data schedules can be significantly more elaborate. In many cases, it is possible for a filter to appear more than once within an input and output data schedule. Figure 3-5b illustrates this possibility.

It is important to note that the input and output data schedules operate in terms of filters rather than cores. The data schedules store filters as sources and destinations rather than cores. The exact core location of each filter is irrelevant.



(a) Simple input and output data schedules



(b) More complex input data schedule

Figure 3-5: Examples of input and output data schedules

3.7 Code Generation

Code generation generates individual source files for each core. Each core source file contains filter code for filters assigned to the core, code to execute the initialization, prologue, and steady-state schedules, and filter communication code. Each core source file contains a single `pthread` thread. The thread executes by first setting core affinity for the thread. The thread then executes the `init` and `prework` methods for all filters assigned to the core, followed by execution of the initialization, prologue and steady-state schedules in that order.

3.7.1 Steady-State Execution Code

In the steady-state, each core simply executes the filters assigned to the core. The number of iterations for stateful filters is determined by the steady-state schedule. The number of iterations for stateless filters is determined by the steady-state schedule

and the number of cores upon which each stateless filter is fissed.

After all filters are executed, each core executes a barrier that spans all cores. The barrier is necessary to keep all cores on the same steady-state execution. The SMP backend presently uses a simple sense-reversing centralized barrier. This barrier is sufficient for shared memory architectures according to [22].

Executing all filters and executing the barrier constitutes a single steady-state iteration. The steady-state is executed either indefinitely or until input to the stream graph runs out.

3.7.2 Filter Communication Code

Filter Input

For each filter in the stream graph, we allocate an input channel. Each input channel consists of multiple buffers linked together in a loop. Multiple buffers are necessary in order to support software pipelining. The first buffer stores input data for the current steady-state iteration. Subsequent buffers store input data for subsequent steady-state iterations. The filter reads from the first buffer while source filters write into the other buffers. Source filters that are n steady-state iterations ahead of the filter write into the $(n+1)^{th}$ buffer.

Because each buffer stores input data for a single steady-state iteration, the size of each buffer is determined by $copydown + multiplicity * pop$ where these are values for the associated filter. The number of buffers in an input channel is determined by one plus the maximum steady-state iteration distance between the filter and its source filters.

Figure 3-6 demonstrates the multiple buffers used in the input channel for filter G. Filter G reads from one buffer, while filters E and F write into the other buffers. Because filter F is one steady-state iteration ahead of filter G, it writes into a buffer that is one away from the buffer filter G reads. Similarly, because filter E is two steady-state iterations ahead of filter G, it writes into a buffer that is two away from the buffer filter G reads. Each buffer stores enough input data for a single steady-state

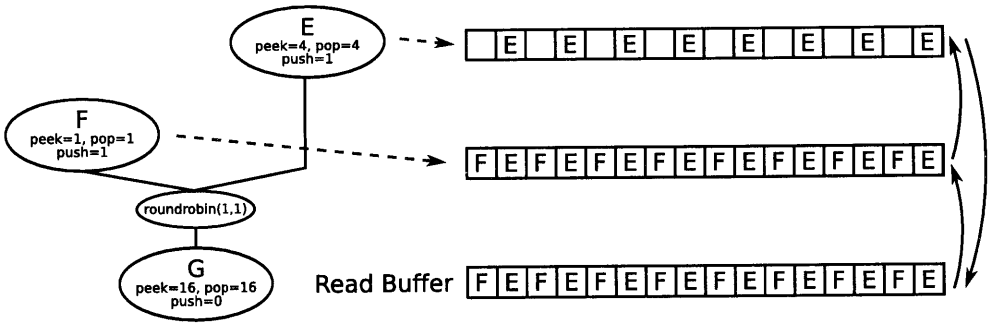


Figure 3-6: Multiple constituent buffers in an input channel

iteration of filter G.

At the end of a steady-state iteration, when a filter has completely processed the input data of the first buffer, the next buffer contains a complete set of new data. The buffers are therefore rotated, giving the filter new data while freeing a buffer to be written into by source filters. To facilitate discussion, the first buffer at any given time is termed the *read buffer* as it is read during steady-state execution.

When buffers are rotated, the last *copydown* elements from the read buffer must be copied to the first *copydown* elements of the next read buffer. These are elements that must be reread by the filter during the next steady-state execution. Because *copydown* elements are copied to the beginning of each new read buffer, source filters avoid writing into the first *copydown* elements of a buffer as these elements will be replaced once the buffer is rotated to the front of the channel.

Filter Output

Filter output is first written to an output buffer. The output data is then distributed from the output buffer to the input buffers of receiving channels. Note that it is imperative for output elements to be written into the correct positions of input buffers. For example, in Figure 3-6, both filters E and F write outputs to their intended positions in filter G's input buffers.

Transferring elements from the output buffer to the input buffers of receiving channels is accomplished through a series of buffer transfers. These buffer transfers are static and never change. As such, these buffer transfer are calculated at compile-

```
g_input[1][0] = output_buffer[0];
g_input[1][2] = output_buffer[1];
g_input[1][4] = output_buffer[2];
g_input[1][6] = output_buffer[3];
              .
              .
              .
```

(a) Buffer transfers for F

```
g_input[2][1] = output_buffer[0];
g_input[2][3] = output_buffer[1];
g_input[2][5] = output_buffer[2];
g_input[2][7] = output_buffer[3];
              .
              .
              .
```

(b) Buffer transfers for E

Figure 3-7: Static buffer transfers for transferring output data

time and hard-coded into the generated source code to execute once the output buffer is full. These buffer transfers are calculated as follows:

1. The output data schedule of the source filter is applied to the filter's output buffer. This helps determine source positions in the output buffer that will be sent to each receiving filter.

2. For each receiving filter, the input buffer that the source filter will write to is determined. The receiving filter's input data schedule is then applied to this input buffer. This helps determine destination positions in the input buffer that expect input from the source filter.

3. Buffer transfers are calculated by iteratively going through the source positions that are sent to each receiving filter and assigning each source position to a destination position expecting input from the source filter.

These static buffer transfers act as static routing for output data. This is a convenient way to conceptualize the process. Figure 3-7 show the buffer transfers for filters E and F that will transfer output from their output buffers to the input buffers of G. Note that the buffer transfers for filters E and F can be easily replaced by a single for-loop. Buffer transfers are looped whenever possible for efficiency.

One important question to consider is why the temporary output buffer is necessary. Why not simply write output directly to the input buffers of receiving channels? This would eliminate the need for the temporary output buffer, which would eliminate an extra copy of each output element. The reason is that the generated code would actually be less efficient. When an output element is generated, table lookups would be

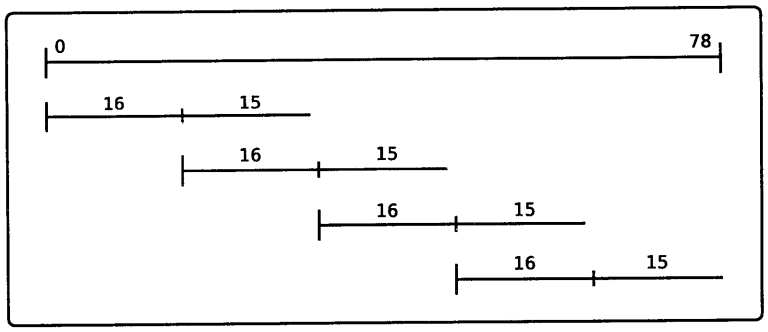


Figure 3-8: Read buffer for Filter B shared by multiple cores

necessary to determine which input buffers receive the element and where the element should be written. These table lookups add run-time overhead to communication.

Writing output to a temporary output buffer and using static buffer transfers eliminates the need for these table lookups since transfers are hardcoded directly into the program. In addition, all elements destined for a single input buffer can be transferred in one consecutive pass. This helps to take advantage of temporal locality in the cache lines of the receiving input buffer. Transferring all elements in one consecutive pass also opens up opportunities for vectorization, resulting in even more efficient code.

3.7.3 Filter Communication Code for Fissed Stateless Filters

Fissed filters require some special treatment in the communication code. Like other filters, a fissed filter receives only one input channel. This one input channel must be shared by the multiple cores executing the fissed filter. To ensure that each core operates on a different part of the filter's input data stream, each core reads a different part of the filter's read buffer. Since the multiplicity of a stateless filter is divided evenly across n cores, each core consumes $(multiplicity/n) * pop$ elements from the filter's read buffer. Core i therefore begins reading the read buffer at the location $(multiplicity/n) * pop * i$.

Figure 3-8 illustrates filter B's input channel shared by four cores. Filter B has a steady-state multiplicity of 64, so each of the four cores receive 16 iterations of filter

B. Each iteration of filter B pops a single data element, so the number of elements consumed by each core is 16. Core i therefore begins reading the read buffer at the location $16 * i$.

Note that for peeking filters, cores need to read more elements than they consume. This is to satisfy the peeking windows of the peeking filters. Each iteration of a peeking filter looks at $peek - pop$ more elements than it consumes. The last iteration of a peeking filter executed by a core therefore requires that the core read $peek - pop$ elements beyond the last elements it consumes. This is illustrated in Figure 3-8. Filter B is a peeking filter where $peek - pop$ is 15. While each core only consumes 16 elements, each core reads an additional 15 elements to satisfy the peeking window of Filter B. Note that this causes data elements to be duplicated on more than one core, which adds some communication overhead.

For the multiple cores executing a fissed stateless filter, each core maintains a separate output buffer for its set of filter iterations. This means that each output buffer contains only a portion of the stateless filter's output data stream. Static buffer transfers are still used to distribute each output buffer to receiving input buffers. However, the above calculations for static buffer transfers must take into account the fact that each output buffer is only a portion of the stateless filter's output data stream. This requires two adjustments:

1. Consider the case where the size of each output buffer is not a multiple of the total weight in the filter's output data schedule. In this case, the start of each output buffer does not always align with the start of the output data schedule. The output data schedule cannot be directly applied to each output buffer. The correct starting position within the output data schedule must first be calculated before it can be applied to an output buffer. For a given core i out of n cores across which a filter is fissed, the number of outputs generated by previous cores is $(multiplicity * push)/n * i$. This can be used to advance the output data schedule, yielding for core i's output buffer the correct starting position within the output data schedule.

2. For receiving filters, the input data schedule can still be applied as usual to determine destination positions that expect input from the fissed stateless filter. However, these destination positions must be shared amongst the multiple output buffers. Core i may need to skip a number of the destination positions if these are written to by previous cores.

3.8 Summary

The end result of compilation is a static scheduling of filters to available cores. Stateless filters are fissed to all cores, while stateful filters are bin-packed across cores. The steady-state schedule generated at compile-time is hard-coded into the code for each core. Cores execute the steady-state schedule in a decentralized manner, using a barrier to keep all cores on the same steady-state iteration. These factors put together statically determine both when and where filters are executed on the cores.

The SMP compiler backend takes advantage of the unique architectural features of commodity multicore machines in a number of ways. Shared memory is used to communicate data between cores. Shared memory is also used to share input data for filters fissed across multiple cores. Layout uses shared caches to minimize communication overhead by placing filters that communicate most with each other on cores that share the same cache. Code generation generates vectorizable code in order to take advantage of instruction-level streaming enchancements.

The partitioning scheme adopted by the SMP compiler backend attempts to statically load-balance the cores, though perfect load-balancing isn't always achieved. Fissing stateless filters to all cores ensures that cores receive the same amount of work. Bin-packing stateful filters also attempts to ensure that cores receive the same amount of work. Unfortunately, this is not always possible. For example, if there are more cores than stateful filters, then some cores will receive less work than others. This is illustrated in Figure 3-4. Inaccurate static work estimates may also cause bin-packing to distribute stateful filters incorrectly to cores. Large load imbalances may result if static work estimates are very inaccurate. The SMP compiler backend

depends on dynamic load-balancing to correct whatever load imbalances may still exist after static load-balancing. Dynamic load-balancing is discussed in Chapter 5.

Chapter 4

Adjusting the Static Schedule at Run-Time

The static schedule generated by the compiler determines how work is distributed on the cores. However, it may be discovered at run-time that the static schedule is non-optimal for run-time conditions. This chapter provides the mechanisms through which the static schedule can be adjusted at run-time to adapt to run-time conditions. These mechanisms allow for stateful and stateless work to be moved between cores as necessary.

Note that adjusting a static schedule to adapt to run-time conditions may be preferable to using fully dynamic scheduling for a number of reasons. In fully dynamic scheduling, the dynamic scheduler must always be continuously running, which adds an enormous run-time overhead. In contrast, the static schedule is only adjusted when needed. If adjustments are not needed, then the static schedule can be used indefinitely without requiring any further calculation. When adjustments are needed, adjustments are inexpensive, having only a small run-time cost. Adjusting a static schedule to adapt to run-time conditions therefore results in considerably less run-time overhead than running a fully dynamic scheduler. In addition, construction of the static schedule at compile-time allows for complex, time-consuming optimizations such as globally minimizing communication. Such optimizations would be too expensive to execute at run-time under fully dynamic scheduling.

num_iters = {16, 16, 16, 16}
start_iter = {0, 16, 32, 48}

(a) Initial schedule

num_iters = {12, 14, 20, 18}
start_iter = {0, 12, 26, 46}

(b) Adjusted schedule

Figure 4-1: Moving stateless filter iterations

4.1 Moving Stateless Work

In the static schedule, iterations of a stateless filter are divided evenly across cores. This section provides the mechanisms necessary to move stateless filter iterations between cores, allowing stateless filter iterations to be arbitrarily distributed across cores.

4.1.1 Implementation Overview

Run-time state is used to store the distribution of stateless filter iterations across cores. For a stateless filter k, two arrays are maintained: num_iters_k and $start\_iter_k$. The first array stores the number of iterations that each core is responsible for executing. The second array stores the starting iteration of each core.

Initially, stateless filter iterations are divided equally amongst cores as dictated by the static schedule. The num_iters_k and $start\_iter_k$ arrays are initialized to reflect this fact. Afterwards, iterations can be moved between cores by updating the two arrays. Figure 4-1 demonstrates the initial setup of the two arrays to reflect the static schedule and the two arrays after iterations have been moved around between cores at run-time.

During steady-state execution, core i executes stateless filter k using $num\_iters_k[i]$ to determine the number of iterations. To ensure that each core executes on different portions of filter k's input data stream, core i begins reading from filter k's read buffer at the location $start\_iter_k[i] * pop_k$. When filter iterations are moved, core i automatically utilizes the new values of $num\_iters_k[i]$ and $start\_iter_k[i]$ to adjust the number of times it executes filter k and where it begins reading filter k's read buffer.

Note that the movement of stateless filter iterations can only occur before steady-state execution and between steady-state iterations. This is because cores depend

on the arrays num_iters_k and $start\_iter_k$ during a steady-state iteration. Modifying these arrays may cause arbitrary, undesired behavior. The only safe times to modify these arrays is therefore before steady-state execution and between steady-state iterations.

4.1.2 Recomputing Buffer Transfers

As described in section 3.7.3, for a fissed stateless filter, each core maintains a separate output buffer for the iterations it executes. Because iterations can be moved between cores, the number of iterations on a core may increase. This may necessitate a resizing of the output buffer in order to accomodate the additional output. A number of strategies may be employed to minimize the number of times the output buffer requires resizing. For simplicity, the size of the output buffer on each core can be set to the maximum number of outputs generated by the stateless filter in a single steady-state iteration. Even if all iterations of a stateless filter were to be moved to a single core, the core would have enough room in its output buffer to store all outputs. This removes any need for resizing at the cost of more memory usage. This is the technique employed by this thesis.

To transfer data from the output buffers to the input buffers of receiving channels, static scheduling makes use of precomputed buffer transfers that copies elements in a fixed pattern. Unfortunately, once iterations are moved between cores, these precomputed buffer transfers are useless since the number of outputs in each output buffer changes. Buffer transfers must therefore be recomputed at run-time whenever iterations are moved.

To allow buffer transfers to be computed at run-time, the brute force solution would be to encode all input and output data schedules into the program. Then whenever filter iterations are moved between cores, the input and output data schedules can be used to compute new buffer transfers for each output buffer of the fissed filter. Unfortunately, this is very expensive and impractical if filter iterations are moved more than once between cores.

Fortunately, recomputing buffer transfers can be made inexpensive if a filter k

meets two constraints. Let w_k^{out} be the total summation of weights in filter k's output data schedule. Let rot_k^{out} be the number of times filter k's output data schedule rotates in a single iteration of filter k, which is equal to $push_k / w_k^{out}$. The first constraint is that rot_k^{out} must be integral.

Consider a filter j that receives data from filter k. Let $w_{k,j}^{out}$ be the total weight of filter j in filter k's output data schedule. Let $w_{j,k}^{in}$ be the total weight of filter k in filter j's input data schedule. Let $rot_{j,k}^{in}$ be the number of times filter j's input data schedule rotates in a single iteration of filter k, which is equal to $rot_k^{out} * w_{k,j}^{out} / w_{j,k}^{in}$. The second constraint is that $rot_{j,k}^{in}$ must be integral. This constraint applies for all filters that receive data from filter k.

If filter k meets the first constraint, each iteration of filter k rotates filter k's output data schedule an integral number of times. Correspondingly, for a given iteration, previous iterations rotate filter k's output data schedule an integral number of times. This means that within an iteration, the first output aligns with the beginning of filter k's output data schedule.

If filter k meets the second constraint for a receiving filter j, each iteration of filter k rotates filter j's input data schedule an integral number of times. Correspondingly, for a given iteration, previous iterations rotate filter j's input data schedule an integral number of times. This means that within an iteration, the first output sent to filter j aligns with the first weight for filter k in j's input data schedule.

Recomputing buffer transfers at run-time for iteration i of filter k can then be accomplished by the following process:

1. Pick a filter j that receives data from filter k

2. Using filter k's output data schedule, calculate the first $rot_k^{out} * w_{k,j}^{out}$ output indices that would normally be sent to filter j. These indices are termed *fixed source offsets.*

3. Using filter j's input data schedule, calculate the first $rot_k^{out} * w_{k,j}^{out}$ input indices that would normally be receiving from filter k. These indices are termed *fixed destination offsets.*

4. Calculate a *source base offset*. Let $core(i)$ be the core that iteration i is assigned to. The source base offset is equal to $(i - start\_iter[core(i)]) * rot_k^{out} * w_k^{out}$.

5. Calculate a *destination base offset*. Let w_j^{in} equal the total weight in filter j's input data schedule. The destination base offset is equal to $i * rot_{j,k}^{in} * w_j^{in}$.

6. Apply the source base offset to the fixed source offsets to get *source indices*.

7. Apply the destination base offset to the fixed destination offsets to get *destination indices*.

8. Iteratively perform buffer transfers between the sources indices and the destination indices.

9. Repeat steps 1-8 for all filters that receives data from filter k

This process for recomputing buffer transfers may at first seem as equally expensive as the brute-force solution. However, note that most of the values in this process are static for all iterations of filter k. The fixed source offsets and fixed destination offsets are all static, as well as the rot_k^{out}, w_k^{out}, $rot_{j,k}^{in}$, and w_j^{in} values. Because these values are static, they can actually be computed at compile-time and hard-coded into the generated source code.

This actually makes recomputing buffer transfers extremely inexpensive. Of course, filter k must first meet the two constraints outlined above. From empirical experience, stateless filters tend to meet these constraints and it is fairly rare to find a stateless filter that does not. Figure 3-5b is an example of where a stateless filter might not meet the above constraints if H, I, or K were to be stateless.

For stateless filters that don't meet the two constraints, it's possible to fallback to the brute-force method to recompute buffer transfers. However, as previously stated, the brute-force method is extremely expensive. In such cases, it might be better to simply not allow filter iterations to be moved.

4.1.3 Alternative to Recomputing Buffer Transfers

One alternative to recomputing buffer transfers at run-time is to simply force all cores running iterations of stateless filter k to write into a centralized output buffer. The centralized output buffer would store all output for filter k. Core i would write output data to the centralized output buffer starting at the location $start\_iter_k[i] * push$. When filter iterations are moved between cores, cores would simply update the positions at which they write into the centralized output buffer. Because the centralized output buffer would not change in size, buffer transfers could be computed at compile-time and would not need to change at run-time despite movement of filter iterations between cores.

The ability to move filter iterations without having to recompute buffer transfers is very attractive. Unfortunately, this approach has a very significant downside. Before buffer transfers can be executed, synchronization would be necessary to ensure that all cores have finished executing their iterations of filter k. This approach essentially adds a centralized joiner after the parallel filter iterations, which would have a negative performance impact.

Because this approach may result in a decrease in performance, it was not implemented during the course of this thesis. However, the ability to move filter iterations without recomputing buffer transfers is still very attractive. Despite the negative performance impact, this approach may be useful in a number of contexts. For example, in cases where brute force recomputation of buffer transfers is required, the centralized buffer approach may be less costly.

4.1.4 Run-Time Cost

The run-time cost of moving stateless filter iterations is typically very low. Updating the two arrays is very inexpensive. Recomputing the buffer transfers is also inexpensive assuming that the optimization detailed above can be applied. If not, the brute force recomputation of buffer transfers is possible, but this is very expensive. In these cases, it might be better to simply disallow the movement of filter iterations. Resizing

output buffers is another potential run-time cost, though a number of strategies can be employed to minimize the number of resizings needed. In cases where significant amounts of memory is available, resizing can be completely avoided by simply making output buffers large enough to store all outputs of a stateless filter in a steady-state execution. Overall, the run-time cost of moving filter iterations is low enough to be practically useful at run-time.

4.2 Moving Stateful Work

In the static schedule, stateful filters are bin-packed across cores. Each stateful filter is simply assigned to a single core. Allowing for the movement of stateful work is fairly trivial. Run-time state is used to store the core assignment of each stateful filter. Each core executes the stateful filters assigned to it according to the run-time state. Moving a stateful filter then simply involves changing the core assignment in the run-time state.

Code generation also needs modification to allow stateful filters to be called from any core. Unfortunately, this change was fairly non-trivial in the StreamIt SMP compiler backend. As such, the capability to move stateful work was not implemented in time for this thesis. This capability will likely be implemented in future work.

4.3 Summary

This chapter contributes methods for adjusting a static schedule at run-time. As previously described, adjusting a static schedule at run-time results in considerably less run-time overhead than fully dynamic scheduling. Seperate methods were developed for moving stateless work and stateful work between cores. Stateless work can be moved in a very fine-grained manner as individual filter iterations can be moved between cores. Stateful work can't be moved in such a fine-grained manner, though moving stateful work is useful if large amounts of work needs to be moved between cores. Moving work between cores is generally inexpensive, which allows the static

schedule to be adjusted more than once during run-time execution. These methods for moving work between cores are utilized by dynamic load-balancing, which will be discussed in the next chapter.

Chapter 5

Dynamic Load-Balancing

The basic goal of dynamic load-balancing is to keep cores load-balanced at all times. Dynamic load-balancing achieves this by periodically adjusting the static schedule, moving work around the cores as necessary to keep the cores load-balanced. Dynamic load-balancing uses the methods outlined in the previous chapter to move work between cores at run-time.

Dynamic load-balancing actively monitors the load on each core. Dynamic load-balancing also actively measures the amount of work in each filter. When load-balancing adjustments are necessary, dynamic load-balancing moves filters using the run-time measurements of filter work to determine which filters to move.

Dynamic load-balancing is capable of fixing load imbalances that might still exist after static load-balancing. As described in section 3.8, load imbalances may arise from inaccurate static work estimates. In the SMP compiler backend, load imbalances may also arise if there are stateful filters. Stateful filters are simply bin-packed across cores, which is an imperfect form of load-balancing that can leave cores imbalanced. In particular, cores are left imbalanced if there are more cores than stateful filters.

Dynamic load-balancing is also capable of responding to real-time events such as sudden changes in core availability. If a core is taken by another process, dynamic load-balancing can shift work away from the core to maintain good performance. Dynamic load-balancing is designed to be as lightweight as possible, adding very little overhead to run-time and adjusting the static schedule only when necessary.

Steady-State Execution Times:

Core 0: 11778966
Core 1: 10808127
Core 2: 10462383
Core 3: 10679319

Filter Execution Times:

	Core 0	Core 1	Core 2	Core 3
Filter 0:	2361276	2360394	2362329	2360691
Filter 1:	1529028	924471	826551	871686
Filter 2:	926676	880542	880623	875223
Filter 3:	928584	865818	821790	875124
Filter 4:	876366	875214	819486	872874
Filter 5:	894159	902592	894429	859671
Filter 6:	888984	897615	847503	897606
Filter 7:	900144	889992	854532	886761
Filter 8:	891936	887427	843543	888453
Filter 9:	888372	888921	834615	890406

Figure 5-1: Example profile of a steady-state iteration

5.1 Run-Time Profiling

Dynamic load-balancing actively monitors the load on each core and the amount of work in each filter. To do so, dynamic load-balancing employs a high-precision timer to make run-time measurements. This thesis makes use of the Time Stamp Counter built into many commodity machines to get direct cycle counts. Use of the Time Stamp Counter isn't imperative, any high-precision timer should be sufficient for the methods described here.

To monitor the load on each core, dynamic load-balancing measures the amount of time it takes for cores to execute a steady-state iteration. Steady-state execution time directly reflects the load on each core. Cores with more work than others will have higher steady-state execution times. Cores that are busy with other processes will similarly have higher steady-state execution times.

Dynamic load-balancing also measures the amount of time each core spends in each filter. This breaks down the steady-state execution times into filter execution times. Together the steady-state execution times and the filter execution times form a steady-state profile. Figure 5-1 illustrates what a steady-state profile might look

like.

To avoid adding too much run-time overhead, steady-state profiles are not generated on every steady-state iteration. Profiles are instead generated at a user-configurable sampling rate. The default sampling rate is once every 10 steady-state iterations. Profiles are stored for later use by the dynamic load-balancer.

5.2 Load-Balancing Algorithm

Load imbalances are detected as differences in steady-state execution times. Dynamic load-balancing moves filters around in the static schedule to make steady-state execution times converge.

Load-balancing is run periodically after a number of steady-state profiles have been generated. The default is to require 10 steady-state profiles, though this is also user-configurable. Steady-state profiles are then averaged together, which involves averaging the steady-state execution times and the filter execution times. Averaging helps to reduce the noise within steady-state profiles.

For each stateless filter, load-balancing calculates the average execution time for a single iteration. This is done by summing the filter execution times for a stateless filter and dividing by the number of iterations executed across cores. Stateless filters are then sorted by average execution time for a single iteration.

Load-balancing uses the steady-state execution times to find the core with the most amount of work and the core with the least amount of work. Let t_{max} be the steady-state execution time of the core with the most work and t_{min} be the steady-state execution time of the core with the least work. Load-balancing attempts to move filter work to cause these two times to converge. Let $t_{transfer}$ be the amount of execution time that load-balancing attempts to move from the core with the most work to the core with the least work. The value $t_{transfer}$ is defined to be $t_{max} - (t_{max} + t_{min})/2$.

To transfer steady-state execution time between the two cores, load-balancing moves iterations of stateless filters. Load-balancing utilizes the sorted list of stateless

filters, starting with the stateless filter with the highest average execution time for a single iteration. As many iterations of this stateless filter is transferred between cores as possible without exceeding $t_{transfer}$. The value $t_{transfer}$ is updated to reflect the amount of execution time that still needs to be transferred. Load-balancing then repeats this process with the next stateless filter in the sorted list. This process is repeated with all stateless filters or until $t_{transfer}$ falls below a user-defined threshold.

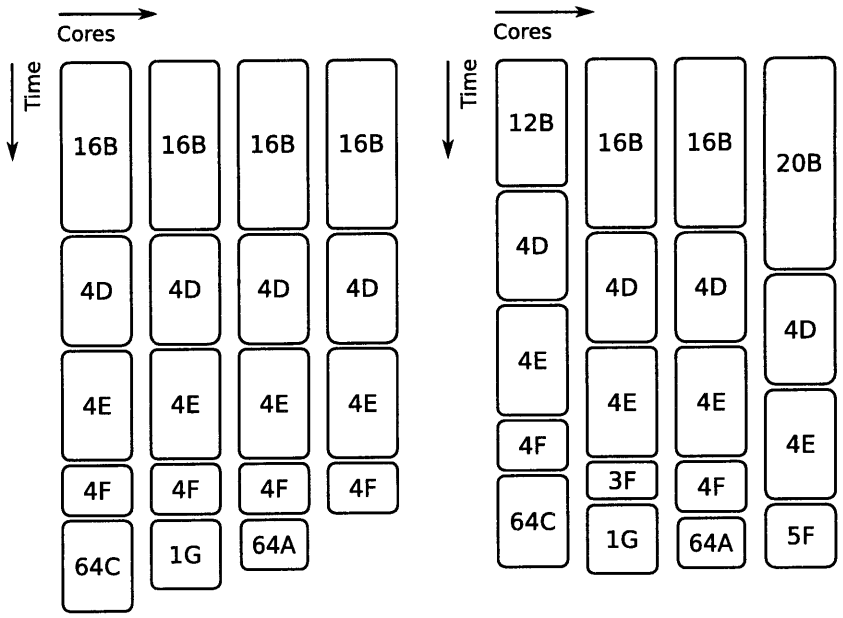
Load-balancing essentially greedily moves iterations of stateless filters, preferentially moving iterations with high average execution times. This helps to minimize the number of filter iterations that are moved between cores, minimizing the amount of changes made to the static schedule.

A single load-balancing pass balances only two cores. Because load-balancing is run periodically, all cores will eventually be load-balanced. The user has the option of allowing multiple load-balancing passes to be run consecutively. This increases the speed at which cores will converge, at the cost of somewhat more expensive load-balancing. Note that multiple consecutive passes actually require slightly less computation than multiple single passes since each consecutive pass beyond the first does not need to re-average steady-state profiles and re-sort stateless filters.

Note that the threshold cutoff for $t_{transfer}$ helps to minimize the cost of load-balancing if cores are already load-balanced. If cores are already load-balanced, then the differences between steady-execution times will be very small, allowing the $t_{transfer}$ cutoff to be applied immediately. Load-balancing therefore adds very little overhead once cores have been balanced.

Sorting the stateless filters may seem very expensive if there are a large number of stateless filters. However, stateless filters remain mostly sorted from one load-balancing pass to the next. Because stateless filters remain mostly sorted, sorting usually requires only a brief pass through the list of filters. An insertion sort is utilized that is optimized for mostly-sorted lists.

Figure 5-2 demonstrates what a static schedule might look like after a few load-balancing passes. After stateless filter iterations are moved, the load imbalance introduced by the stateful filters is largely nullified. A tiny amount of load imbalance still



(a) Before Load-Balancing (b) After Load-Balancing

Figure 5-2: Example of a static schedule after load-balancing

exists, but this is fairly negligible. In the above diagram, load-balancing results in an 8% increase in performance. The amount of improvement would be greater if the stateful filters were to contain more work. The performance impact of load-balancing on actual benchmarks is detailed in section 7.4.1.

Note that the current load-balancing algorithm is presently unable to move stateful filters. This is because the capability to move stateful filters was not fully implemented as discussed in section 4.2. This has an unfortunate effect on load-balancing. If all stateless filter iterations have been moved away from a core and only stateful filters are left, load-balancing can't move any further work away from the core.

Cores utilized by other processes should have high steady-state execution times, which means that load-balancing should automatially handle these cases by moving stateless filter iterations away from these cores. If there are any stateful filters on the core, however, the load-balancer is presently out of luck.

5.3 Enhancements to Load-Balancing

A number of possible enhancements can be made to this load-balancing scheme. One particular enhancement would allow load-balancing to respond more quickly to changes in core availability. When a steady-state profile is taken, if a core experiences a dramatic increase in steady-state execution time, then triggering immediate load-balancing would be appropriate. This immediate load-balancing could perform more load-balancing passes than usual to encourage rapid convergence. This enhancement might significantly improve the response of dynamic load-balancing to changes in core availability.

An important enhancement would be to allow stateful filters to be moved between cores. This capability was not fully implemented at the time of this thesis. Extending the load-balancing algorithm to move stateful filters would be fairly trivial. The current algorithm sorts stateless filters by the average execution time of a single iteration. Stateful filters can be inserted into this sorted list by treating stateful filters as a single iteration that must be moved all at once.

It would be fairly useful to consider a load-balancing algorithm that would balance more than a pair of cores at a time. This would improve the convergence time of load-balancing. However, a more complex load-balancing algorithm will also make load-balancing more expensive. The additional overhead might not be worth it. Note that balancing more than a pair of cores at a time can already be emulated by running more than one load-balancing pass consecutively. This option can be turned on by the user if desired.

Special behavior at the start of a stream program might help with rapid initial convergence. For example, running extra load-balancing passes at the beginning of a program. Also, if a stream program is compiled for 16 cores and a machine only contains 4 cores, a method for immediately redistributing work would be very useful. This could be done very simply for stateless filters by simply redistributing iterations evenly across the available cores. A special algorithm could then be implemented to redistribute stateful filters.

5.4 Summary

This chapter contributes a dynamic load-balancing algorithm that adjusts the static schedule at run-time to adapt to run-time conditions. Dynamic load-balancing attempts to fix load imbalances that might exist after static scheduling and to compensate for cores that are shared with other processes by moving work away from these cores. Dynamic load-balancing also attempts to be inexpensive, adding very little overhead to run-time execution. While a large number of enhancements are possible, the current dynamic load-balncing algorithm should be effective at accomplishing these goals.

Chapter 6

Optimizing Filter Fission for Reduced Communication

As previously described, filter fission is a process that takes the iterations of a stateless filter and divides them evenly across multiple cores. This allows the iterations of a stateless filter to be run in parallel, taking advantage of the data parallelism inherent to stateless filters. Unfortunately, filter fission introduces significant communication overhead. First, data must be distributed to and from cores. Second, fission of peeking filters requires input data to be duplicated to multiple cores, in some cases requiring input data to be duplicated to all cores. This communication overhead can be a significant bottleneck that slows program execution.

This chapter describes two optimizations that attempt to minimize the communication overhead introduced by filter fission. The first optimization reduces the percentage of input data that must be duplicated to multiple cores when peeking filters are fissed. The second optimization exploits locality to reduce the percentage of data that must be transmitted between cores when pipelines of stateless filters are fissed.

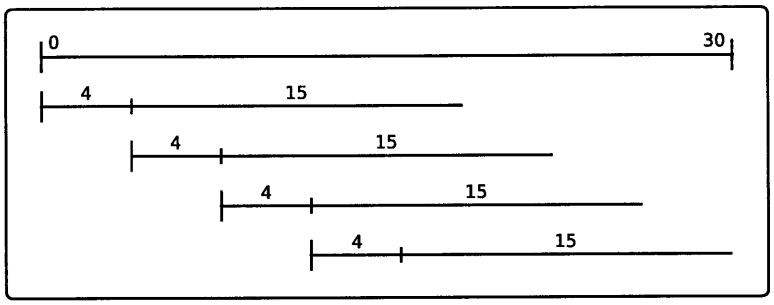


Figure 6-1: Data duplication as a result of peeking

6.1 Reducing Data Duplication

When a stateless filter is fissed across multiple cores, fission gives each core a disjoint set of input data to consume. Unfortunately, peeking complicates this by requiring cores to view more data than they consume. While the input data consumed by each core is disjoint, the input data that each core must view as a result of peeking is not disjoint. Because the input data that each core must view is not disjoint, some of the input data must be duplicated to multiple cores.

For example, consider a peeking filter that has a *pop* rate of 1 and a *peek* rate of 16. Suppose that the peeking filter has a steady-state multiplicity of 16 and that the filter is fissed across four cores, giving each core four iterations of the peeking filter. Figure 6-1 illustrates the read buffer for this peeking filter, the input elements that each core consumes, and the input elements that each core must view as a result of peeking. While each core consumes only four input elements, each core must view an additional 15 elements to satisfy the peeking window of the filter. As a result of peeking, most of the input elements must be duplicated to at least two cores. A significant amount of the input elements must be duplicated to all four cores.

Duplicating input elements to multiple cores is undesirable as it comes with a number of performance costs. The cache lines associated with the input elements must be replicated to multiple cores, which requires time and communication. When the cache lines later receive writes from source filters, the replicated cache lines must either be invalidated or updated (depending on the cache coherency protocol), which also requires time and communication. If cache lines are replicated across many

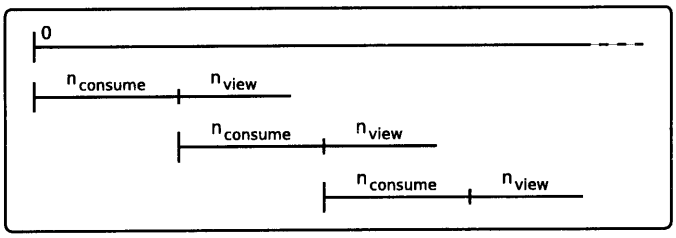


Figure 6-2: Data duplication between neighboring cores

cores, the cost of cache line invalidation/update can be very significant. Reducing the amount of input elements that are duplicated to multiple cores is desirable as it helps avoid these performance costs.

For a peeking filter k that is fissed evenly across n cores, an optimization can be applied to reduce the percent of input data that must be duplicated to multiple cores. Let $n_{consume}$ be the number of input elements that a single core consumes from the read buffer. Let n_{view} be the number of elements that a single core must view beyond the elements it consumes. These values are defined to be:

$$n_{consume} = (multiplicity_k/n) * pop_k \tag{6.1}$$

$$n_{view} = peek_k - pop_k \tag{6.2}$$

We can increase $n_{consume}$ by increasing the steady-state multiplicity of filter k. If we increase the steady-state multiplicity of filter k such that $n_{consume}$ is greater than n_{view}, this guarantees that input data must only be duplicated between neighboring cores. This is illustrated in Figure 6-2. We can then calculate the percent of input data that must be duplicated between neighboring cores as:

$$\%\ of\ duplication = 100 * \frac{n_{view}}{n_{consume}} \tag{6.3}$$

Given this equation, the percent of input data that must be duplicated between neighboring cores can be decreased by further increasing the steady-state multiplicity of filter k. The compiler exploits this to decrease for filter k the percent of input data

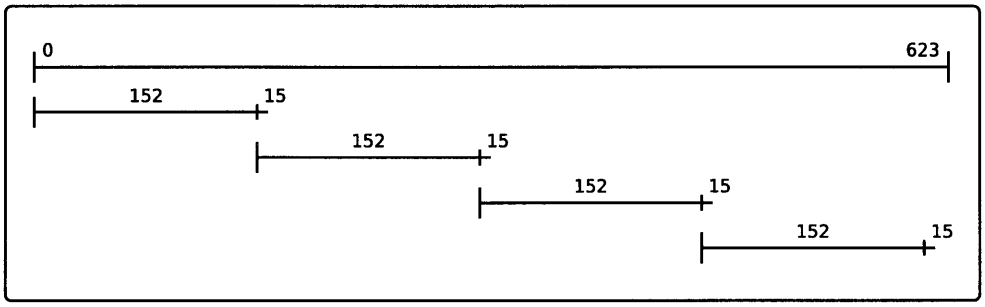


Figure 6-3: Reduced data duplication after optimization

that must be duplicated between cores to less than 10%. This requires increasing the steady-state multiplicity of filter k to meet the following constraint:

$$\begin{aligned} \%\ of\ duplication &\leq 10 \\ 100 * \frac{n_{view}}{n_{consume}} &\leq 10 \\ 100 * \frac{peek_k - pop_k}{(multiplicity_k/n) * pop_k} &\leq 10 \\ multiplicity_k &\geq 10 * \frac{(peek_k - pop_k) * n}{pop_k} \end{aligned} \tag{6.4}$$

Decreasing the percent of input data that must be duplicated between cores optimizes fission for filter k as this helps to decrease the communication costs of fission. This optimization is applied to all fissed peeking filters in the stream graph. Applying this optimization to the example in Figure 6-1 yields Figure 6-3. The steady-state multiplicity of the peeking filter has been increased by a factor of 38 to 608. The amount of input duplication has decreased dramatically to 9.9%.

Note that to increase the steady-state multiplicity of a filter, the steady-state multiplicities for all filters in the stream graph must be multiplied by a constant integral factor. This is necessary to maintain the property that execution of the steady-state schedule does not change the number of elements buffered on each channel. Applying this optimization therefore affects the entire stream graph.

Because all steady-state multiplicities must be increased simultaneously, this op-

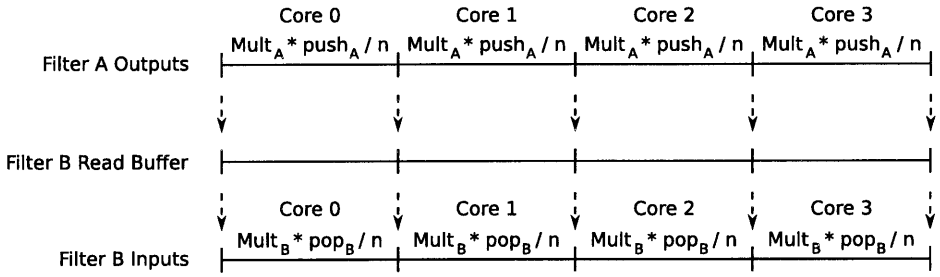


Figure 6-4: Exploiting locality when receiving filter has no copydown

timization has effects other than decreasing the percent of input data data that must be duplicated. Buffering requirements are increased as buffers must store more data per steady-state. Latency of the stream program is also increased as each steady-state iteration lasts longer and results may not be available until the end of a steady-state. It is assumed that the increased buffering is acceptable for SMP machines and that the increased latency is acceptable for affected stream applications.

6.2 Exploiting Locality

Consider a pipeline that contains the stateless filters A and B, where filter A is the source filter for filter B. Suppose that filters A and B are fissed onto the same n cores. Ideally, we would like to arrange the iterations of A and B such that on a given core, the output produced by iterations of filter A can be used by iterations of filter B on the same core. This would remove the need to send filter A's output to other cores, which would reduce the amount of data that needs to be transmitted between cores.

To achieve the above goal, iterations of filters A and B are assigned to cores using a simple algorithm. Let $mult_A$ be the steady-state multiplicity of filter A and $mult_B$ be the steady-state multiplicity of filter B. Core 0 receives the first $mult_A/n$ iterations of filter A and the first $mult_B/n$ iterations of filter B. Core 1 receives the next $mult_A/n$ iterations of filter A and the next $mult_B/n$ iterations of filter B. This is repeated for subsequent cores.

To see why this achieves the above goal, consider the case where filter B has a *copydown* of zero. In this case, the output of filter A is written to filter B's read buffer

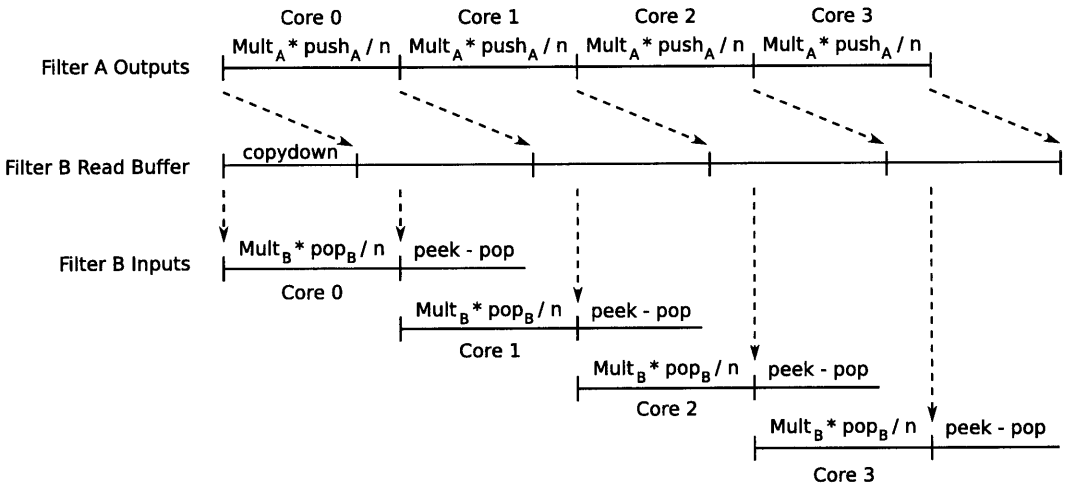


Figure 6-5: Exploiting locality when receiving filter has copydown

with no offset. Given that filter A writes to filter B's read buffer with no offset, and given how iterations of filters A and B are distributed to cores, it is easy to see that the outputs of filter A on a core perfectly match the inputs of filter B on the same core. This is illustrated in Figure 6-4. Each core produces $mult_A * push_A/n$ elements for filter A and consumes $mult_B * pop_B/n$ elements for filter B. These two values are equal due to the way that the steady-state schedule is constructed. As illustrated in Figure 6-4, the output that a core generates for filter A is consumed on the same core and do not need to be transmitted to other cores. This setup therefore results in no communication between cores. The above algorithm for distributing iterations of filter A and B therefore achieves the stated goal of exploiting locality to reduce the amount of data transmitted between cores.

Unfortunately, this breaks down when filter B has a non-zero *copydown*. In this case, the output of filter A must be written to filter B's read buffer with an offset of *copydown*. Given how the iterations of filters A and B are distributed to cores, the outputs of filter A on a core no longer perfectly match the inputs of filter B on the same core. This is illustrated in Figure 6-5. As a result of *copydown*, only some of the output that a core generates for filter A can be used by the iterations of filter B on the same core. The rest of the output for filter A must be transmitted to subsequent cores.

In fact, if *copydown* is larger than $mult_B * pop_B/n$, then none of the output that a core generates for filter A can be used by iterations of filter B on the same core. Allowing *copydown* to be larger than $mult_B * popB/n$ therefore removes all locality, requiring that all of the output that a core generates for filter A to be transmitted to other cores. This results in a significant amount of data that must be transmitted between cores, which is what we would like to avoid.

To restore locality, we can apply the constraint that *copydown* must be less than $mult_B * pop_B/n$. If filter B does not meet this constraint, we can increase its steady-state multiplicity until this constraint is satisfied. Satisfying this constraint guarantees that at least some of the output that a core generates for filter A can be used by iterations of filter B on the same core.

Given that filter B meets the above constraint, the percent of filter A's output that can be used by iterations of filter B on the same core is equal to:

$$\%\ of\ locality = 100 * (1 - \frac{copydown}{mult_A * push_A/n}) \tag{6.5}$$

Correspondingly, the percent of filter A's output that must be transmitted to other cores is equal to:

$$\%\ transmitted = 100 * \frac{copydown}{mult_A * push_A/n} \tag{6.6}$$

We can increase the steady-state multiplicity of filter A to improve locality and decrease the percent of filter A's output that must be transmitted between cores. This helps to reduce the amount of communication between cores. The compiler presently aims to decrease the percent of output transmission to be less than 10%.

As discussed in the previous section, to increase the steady-state multiplicity of a filter, the steady-state multiplicities for all filters in the stream graph must be multiplied by a constant integral factor. This optimization therefore effects the entire stream graph. As discussed in the previous section, increasing steady-state multiplicities for all filters in the stream graph results in increased buffering requirements and increased latency.

6.3 Summary

This chapter contributes optimizations to filter fission that help to reduce the communication overhead associated with filter fission. These optimizations reduce the percentage of input data that must be duplicated between cores when peeking filters are fissed, and the percentage of data that must transmitted between cores when pipelines of stateless filters are fissed. These optimizations operate by increasing the steady-state multiplicities of filters being fissed. Note that these optimizations affect the entire stream graph as increasing the steady-state multiplicity of a filter requires increasing the steady-state multiplicities of all filters.

Chapter 7

Performance Evaluation

This thesis contributes two elements designed to improve the performance of stream programs on SMP machines: optimized filter fission and dynamic load-balancing to perform dynamic adjustments to the static schedule. This chapter characterizes the performance of these improvements.

7.1 Evaluation Setup

Performance was evaluated on a commodity 16-core machine built from Intel Xeon E7340 processors. These are 64-bit quad-core processors that each consists of two dual-core dies, where each die contains a 4 MB L2 cache shared across both cores. The front side bus runs at 1066 MHz and the machines have 16 GB of main memory.

A number of previously developed StreamIt benchmarks were used during the course of this performance evaluation. A brief description of these benchmarks is provided in Table 7.1. Table 7.2 provides a characterization of these benchmarks after the benchmarks have been partitioned across 16 cores.

The source code generated by the StreamIt compiler for these benchmarks was compiled using the Intel C++ Compiler, version 11.1. This compiler contains powerful vectorization optimizations that allow programs to take advantage of instruction-level streaming enhancements. These vectorization optimizations are very applicable for stream programs. Benchmarks were compiled using the -O2 optimization level.

Benchmark	Description	Lines of Code
Radar (fine)	Radar array front end with fine-grained filters	201
Audiobeam	Audio beamformer, steers channels into a single beam	167
ChannelVocoder	Channel voice coder	135
Filterbank	Filter bank for multi-rate signal processing	134
TargetDetect	Target detection using matched filters and threshold	127
FMRadio	FM radio with equalizer	121
FFT (coarse)	64-point FFT (coarse-grained)	116
FFT (medium)	64-point FFT (medium-grained)	53
RateConvert	Audio down-sampler	58
TDE	Time-delay equalization	102
MatrixMult (fine)	Fine-grained matrix multiply	79
MatrixMult (coarse)	Blocked-matrix multiply	120

Table 7.1: Benchmarks used in performance evaluation

Benchmark	# of stateless filter instances	# of stateful filter instances	% stateless work	% stateful work
Radar (fine)	8	16	1.52 %	98.48 %
Audiobeam	15	1	57.58 %	42.42 %
ChannelVocoder	34	1	99.96 %	0.04 %
Filterbank	24	1	99.88 %	0.12 %
TargetDetect	4	1	97.72 %	2.28 %
FMRadio	3	0	100.00 %	0.00 %
FFT (coarse)	1	0	100.00 %	0.00 %
FFT (medium)	1	1	91.82 %	8.18 %
RateConvert	1	2	97.72 %	2.28 %
TDE	1	0	100.00 %	0.00 %
MatrixMult (fine)	1	1	95.68 %	4.32 %
MatrixMult (coarse)	2	2	83.86 %	16.14 %

Table 7.2: Benchmarks characteristics after partitioning to 16 cores

7.2 SMP Compiler Backend Performance

A brief performance characterization of the SMP compiler backend is given here. This characterization does not include the effect of optimized filter fission and dynamic load-balancing and will therefore act as a performance baseline for these features.

The SMP compiler backend was evaluated for throughput and parallel speedup on the benchmarks outlined in Table 7.1. Figure 7-1 graphically illustrates the parallel speedups achieved by the SMP compiler backend, while Table 7.3 provides exact throughput and speedup values. Throughput values are provided in terms of number of outputs per 10^5 processor cycles. On 16 cores, the ideal speedup for each benchmark is 16x. Unfortunately, the ideal throughput for each benchmark is largely unknown as many of the benchmarks have never been executed on the hardware used in this performance evaluation.

The performance of the SMP backend is largely subpar, with only a handful of the benchmarks achieving any meaningful speedup. An analysis of execution costs shows that the steady-state barrier consumes a surprisingly large amount of the execution time. Figure 7-2 gives a breakdown of execution time in terms of time spent executing useful work and time spent in the steady-state barrier. The amount of time spent in the steady-state barrier ranges from 16% to 79%.

The best-performing benchmarks spend the least amount of time in the steady-state barrier. These benchmarks contain large amounts of steady-state work, allowing them to better amortize the cost of the steady-state barrier. For smaller benchmarks, the barrier adds a significant overhead that impedes parallel performance. Table 7.3 provides static estimates for the amount of steady-state work contained in each benchmark. Benchmarks with larger static work estimates generally exhibit better speedup values.

Unfortunately, the barrier does not fully explain the non-ideal speedups of the SMP backend. Figure 7-3 illustrates the theoretical speedups that would be achieved if the steady-state barrier had absolutely no run-time cost. While speedups would be significantly improved, many of the benchmarks would still fail to achieve anywhere

Benchmark	1 Core Throughput	16 Core Throughput	Parallel Speedup	Steady-State Work	Comp/Comm Ratio
Radar (fine)	10.89	61.88	5.68	3032192	1633.72
Audiobeam	68.03	81.43	1.2	60160	10.9
ChannelVocoder	7.01	46.6	6.65	18950736	304.01
Filterbank	10.58	80.13	7.57	6075776	114.65
TargetDetect	55.19	93.28	1.69	212240	11.02
FMRadio	34.84	41.95	1.2	347888	83.95
FFT (coarse)	2947.37	11673.26	3.96	469536	114.63
FFT (medium)	2078.66	1648.96	0.79	148912	72.71
RateConvert	48.0	85.1	1.78	324944	65.51
TDE	674.17	9671.05	14.6	7301904	211.28
MatrixMult (fine)	677.19	1811.07	2.93	1082768	225.58
MatrixMult (coarse)	791.28	1454.81	1.98	891312	70.34

Table 7.3: Throughput and speedup values for SMP Backend

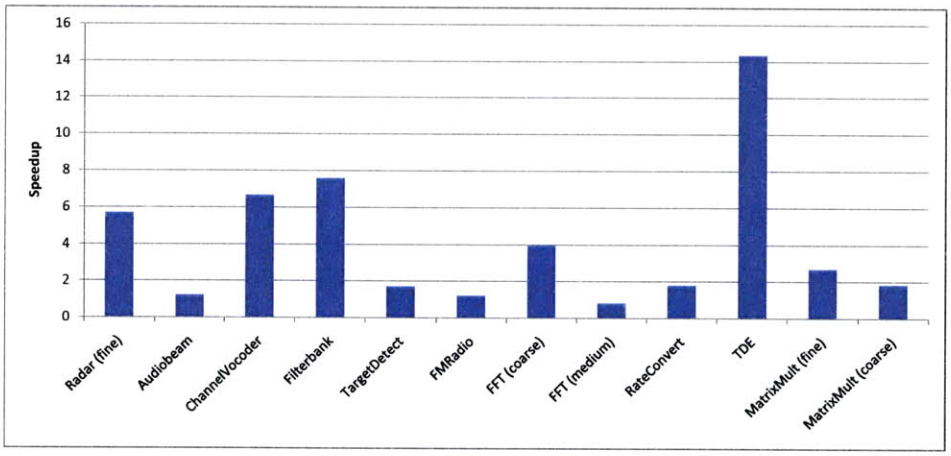


Figure 7-1: Speedup achieved by SMP Backend

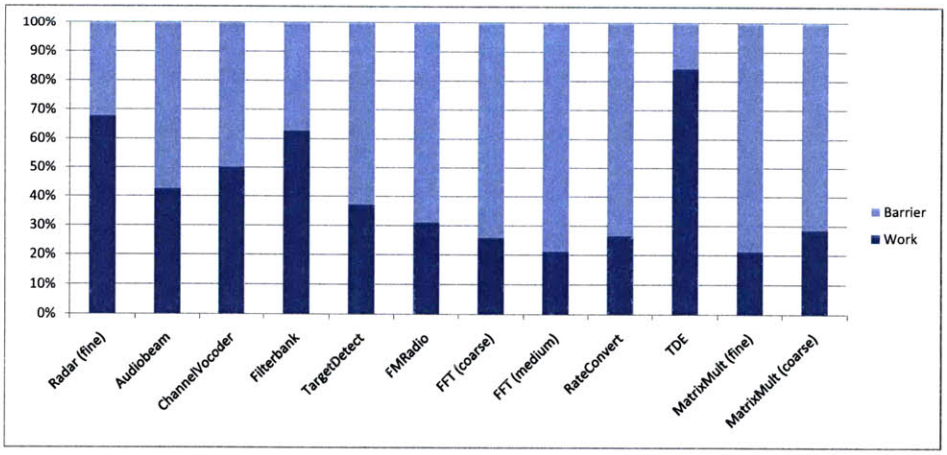


Figure 7-2: Breakdown of execution costs

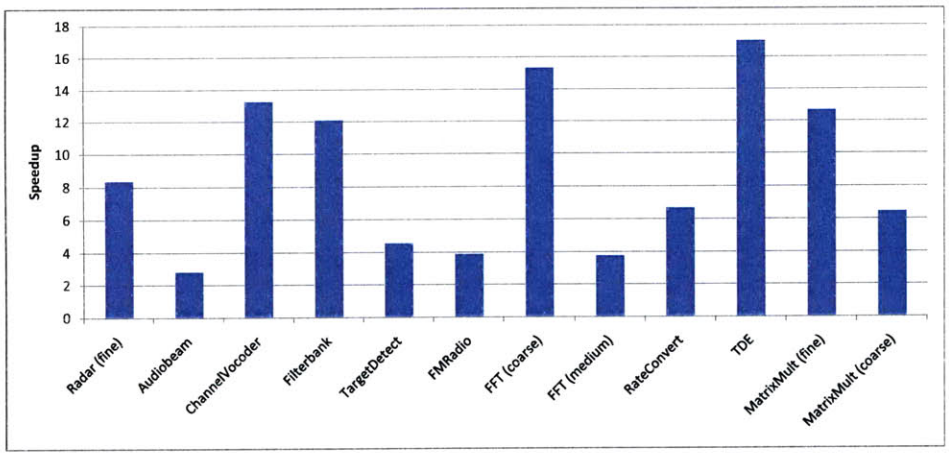


Figure 7-3: Theoretical speedup without barrier

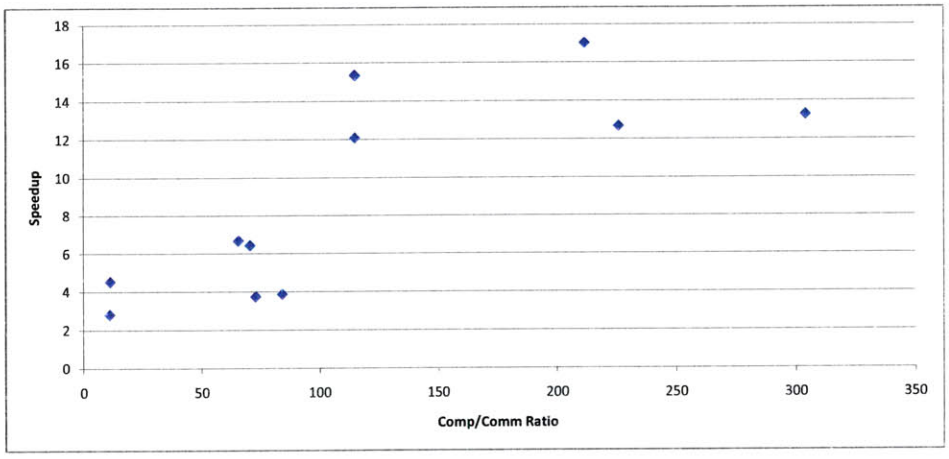


Figure 7-4: Theoretical speedup versus computation/communication ratio

near 16x speedup.

Communication is believed to be one of the major overheads that prevent ideal scaling. To explore this, computation/communication ratios were calculated for the benchmarks. Computation/communication ratios are simply ratios between static estimates of work and communication. These ratios are provided in Table 7.3. Figure 7-4 graphs the theoretical speedup values against the computation/communication ratios. This graph shows a rough correlation between theoretical speedup values and computation/communication ratios. As computation increases relative to communication, the theoretical speedup achieved by the SMP backend improves. This suggests

that communication does indeed have a significant impact on the achieved speedups.

7.3 Optimized Filter Fission

The goal of optimized filter fission is to decrease the communication overhead associated with fission. In particular, optimized filter fission decreases the amount of data duplication that results from fissing peeking filters. Optimized filter fission also decreases the amount of data that must be transmitted between cores for fissed pipelines of stateless filters.

To evaluate the effect of optimized filter fission, new performance numbers were collected with optimized filter fission turned on. Note that for many of the benchmarks, optimized filter fission did not have any effect on the stream graph. This section focuses primarily on the benchmarks for which optimized filter fission did have an effect.

Table 7.4 provides new throughput and speedup values with optimized filter fission turned on. Figure 7-5 graphically illustrates the effect of optimized filter fission on speedup values. Optimized filter fission appears to have an enormous impact on the affected benchmarks. With optimized filter fission turned on, most of the affected benchmarks exhibit roughly 16x speedups.

The TargetDetect benchmark is a particularly interesting case that appears to achieve 43.7x speedup with optimized filter fission turned on. A close examination of the benchmark yields that single core execution is particularly slow. In the single core case, an optimization that would fuse the benchmark to a single filter is missed. Forcing the compiler to fuse the benchmark to a single filter causes a dramatic increase in single core performance. With the improved single core performance, the speedup of the TargetDetect benchmark is reduced to a more reasonable 15.7x.

While the intended effect of optimized filter fission is to decrease the communication overhead introduced by the fission, optimized filter fission also increases the steady-state multiplicities for all filters in each affected benchmark. This results in more steady-state work for each affected benchmark, allowing each benchmark to

Benchmark	1 Core Throughput	16 Core Throughput	Parallel Speedup	Multiplicity Factor	Steady-State Work	Comp/Comm Ratio
Audiobeam	68.03	84.13	17.47	2080	7820800	15.12
ChannelVocoder	7.01	950.25	15.01	208	246359568	613.45
Filterbank	10.58	463.37	20.39	2544	966048384	1341.02
TargetDetect	55.19	41.47	43.7	47840	634597600	1579.17
FMRadio	34.84	156.79	18.3	20320	441817760	2652.35
RateConvert	48.0	189.33	11	7984	162147056	1750.9

Table 7.4: Throughput and speedup values with optimized filter fission

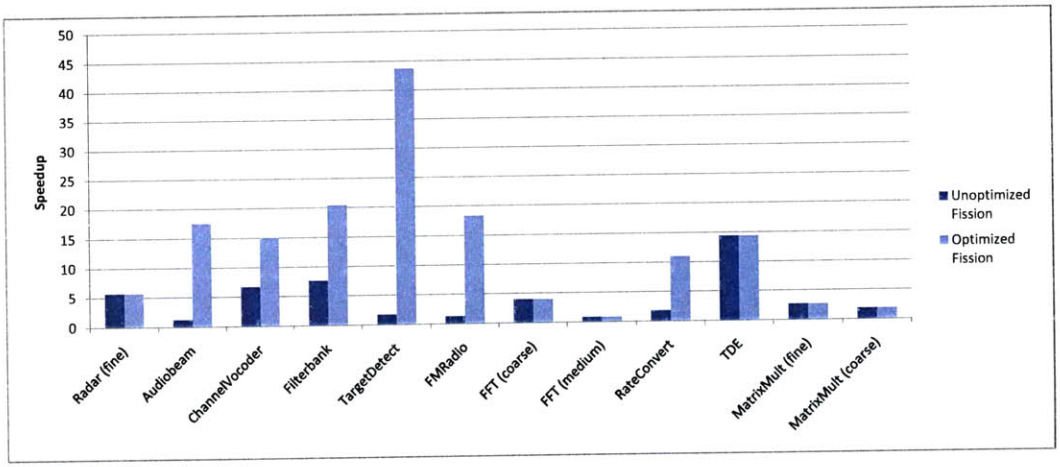


Figure 7-5: Speedup with optimized filter fission

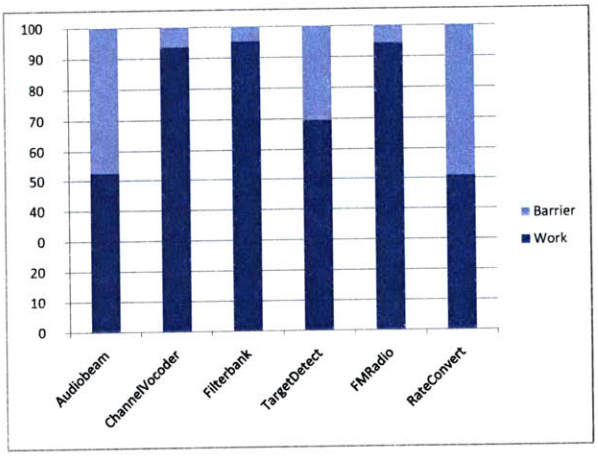


Figure 7-6: Breakdown of execution costs with optimized filter fission

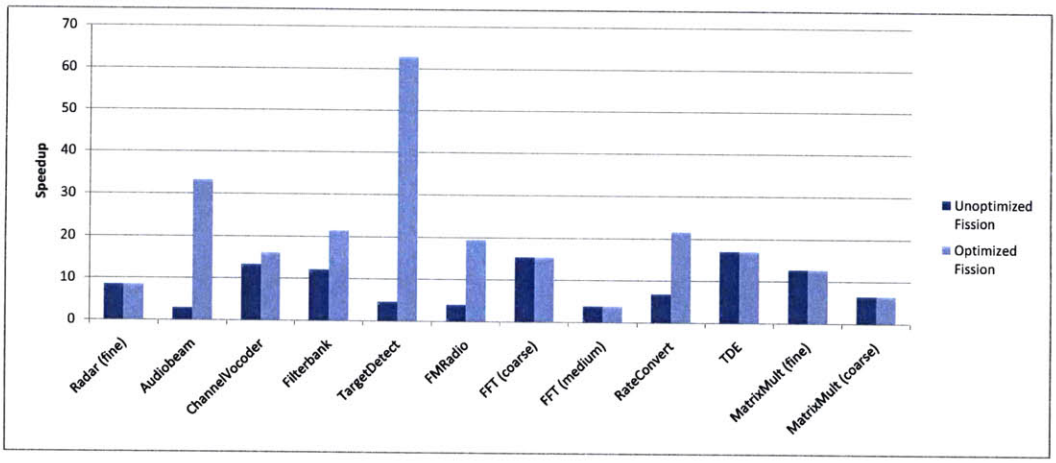


Figure 7-7: Effect of optimized filter fission on theoretical speedup without barrier

better amortize the run-time cost of the steady-state barrier. This may help explain why benchmarks achieve much better scaling with optimized filter fission turned on.

Table 7.4 contains the constant multiplicity factors used to scale up steady-state multiplicities in each affected benchmark. Figure 7-6 gives an updated breakdown of execution time in terms of time spent executing useful work and time spent in the steady-state barrier. As can be seen in most of the affected benchmarks, the increase in the steady-state multiplicities results in the steady-state barrier consuming a much smaller percentage of execution time.

To differentiate between speedup caused by decreased communication overhead and speedup caused by better amortization of the steady-state barrier, we again calculate the theoretical speedups that would be achieved if the steady-state barrier had absolutely no run-time cost. This factors out the speedup caused by better amortization of the steady-state barrier. Figure 7-7 illustrates the theoretical speedups with and without optimized filter fission. We can see that theoretical speedups dramatically improve with optimized filter fission turned on. This means that optimized filter fission results in significantly reduced communication overhead, which in turn results in more efficient program execution.

7.4 Dynamic Load-Balancing

Dynamic load-balancing has three basic goals: fixing inaccurate static load-balancing, dynamically adjusting to changes in core availability, and adding very little overhead to run-time execution. This chapter evaluates the performance of dynamic load-balancing in these three areas.

Note that the default parameters for dynamic load-balancing are used in this characterization. Run-time profiling is performed once every 10 steady-state iterations and load-balancing is performed once every 10 run-time profiles. Load-balancing performs a single load-balancing pass, which load-balances a single pair of cores.

7.4.1 Fixing Inaccurate Static Load-Balancing

To determine how well dynamic load-balancing can fix inaccurate static load-balancing, the performance of each benchmark is measured with dynamic load-balancing turned on. This is then compared to the previous numbers in this chapter as these numbers reflect the performance of the static schedule. Ideally, dynamic load-balancing should be able to adjust the static schedule to yield a net positive gain in performance.

Figure 7-8 compares the speedups achieved by the static schedule to the speedups that can be achieved with dynamic load-balancing turned on. Figure 7-9 shows the performance impact of dynamic load-balancing in terms of percentages. Note that these numbers are collected with optimized filter fission turned on.

For most of the benchmarks, dynamic load-balancing results in a small performance gain ranging from 2% to 13%. These are benchmarks that are already mostly load-balanced after compile-time. In these cases, dynamic load-balancing makes very few adjustments to the existing static schedule, usually only moving a few filter iterations before achieving a relatively steady load-balanced state. Note that this demonstrates the capability for the dynamic load-balancing system to make very fine-grained adjustments.

TargetDetect and RateConvert receive substantial performance improvements once dynamic load-balancing is turned on. These benchmarks contain severely imbalanced

Benchmark	1 Core Throughput	16 Core Throughput	Parallel Speedup	Steady-State Work	Comp/Comm Ratio
Radar (fine)	10.89	64.47	5.92	3032192	1633.72
Audiobeam	68.03	1279.43	18.81	7820800	15.12
ChannelVocoder	7.01	113.41	16.18	246359568	613.45
Filterbank	10.58	229.91	21.73	966048384	1341.02
TargetDetect	55.19	3267.97	59.22	634597600	1579.17
FMRadio	34.84	652.66	18.73	441817760	2652.35
FFT (coarse)	2947.37	12135.92	4.12	469536	114.63
FFT (medium)	2078.66	1732.5	0.83	148912	72.71
RateConvert	48.0	886.05	18.46	162147056	1750.9
TDE	674.17	9560.23	14.18	7301904	211.28
MatrixMult (fine)	677.19	2044.54	3.02	1082768	225.58
MatrixMult (coarse)	791.28	1168.04	1.48	891312	70.34

Table 7.5: Throughput and speedup values with dynamic load-balancing

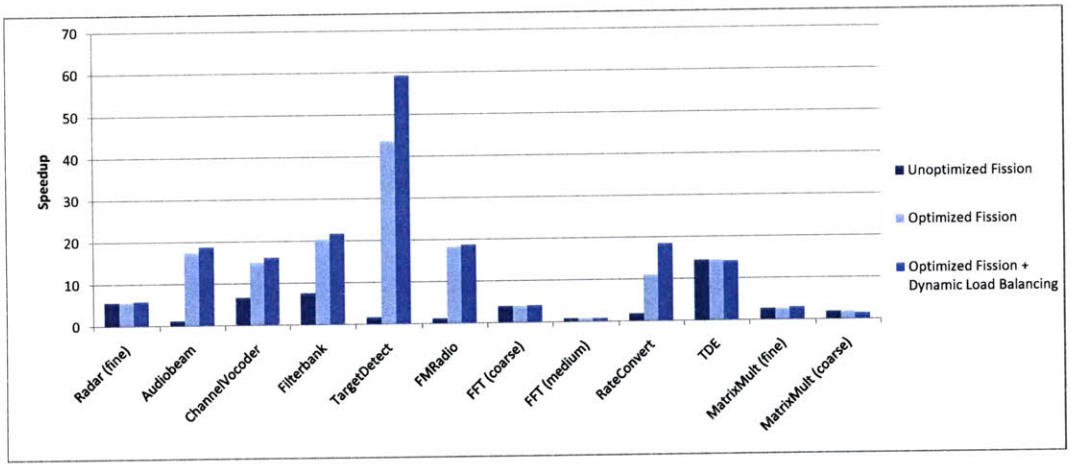


Figure 7-8: Speedup with dynamic load-balancing

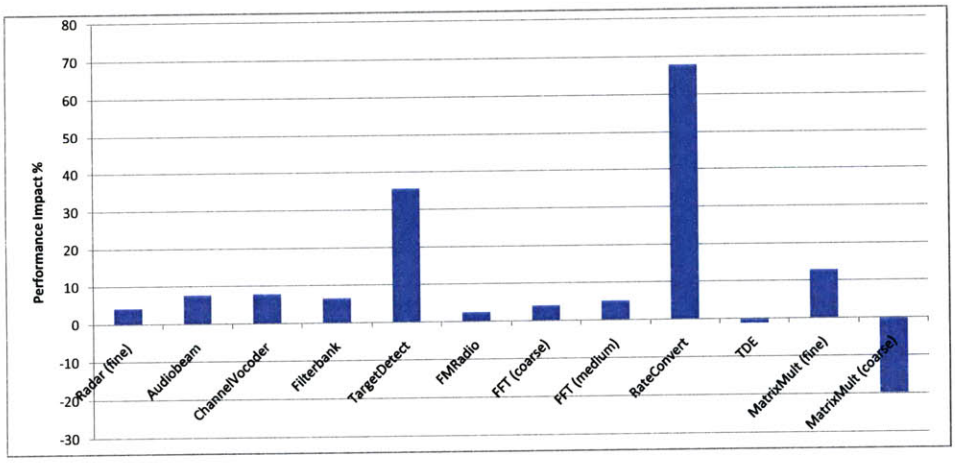


Figure 7-9: Performance impact of dynamic load-balancing

loads after static scheduling. Both benchmarks contain a large stateful filter assigned to the first core and a number of stateless filters fissed across all cores. The large stateful filter causes the first core to have considerably more work than the other cores. As a result, the first core is a bottleneck at run-time. Dynamic load-balancing quickly moves stateless filter iterations away from the first core until the cores are load-balanced. Load-balancing converges fairly rapidly, with the bulk of load-balancing happening within a few load-balancing passes.

In contrast, coarse MatrixMult receives substantial performance degradation once dynamic load-balancing is turned on. Close examination reveals that the code generated with dynamic load-balancing cannot be as optimized by the Intel C++ Compiler than the code generated without dynamic load-balancing. Within the benchmark, one of the stateless filters is fissed such that each core receives a single iteration of the filter. With dynamic load-balancing turned off, it is statically known that each core will execute only one iteration of the filter. As a result, the Intel C++ Compiler can remove the iteration loop around the filter, which greatly improves the performance of the compiled code.

With dynamic load-balancing turned on, it is not statically known that each core will execute only one iteration of the filter. This is because filter iterations can be moved between cores. As such, the iteration loop around the filter cannot be removed by the Intel C++ Compiler. Dynamic load-balancing therefore removes an opportunity for optimization within the generated code, which is what results in performance degradation.

Overall, dynamic load-balancing is very successful at fixing load imbalances that remain after static load-balancing. As demonstrated by the benchmarks, dynamic load-balancing is capable of making both fine-grained and large-scale adjustments to the static schedule. The capability for fine-grained adjustments allows for performance gains even in cases where the static schedule is already mostly load-balanced.

Benchmark	No Load w/ Load-Balancing	Loaded w/o Load-Balancing	Loaded w/ Load-Balancing
Radar (fine)	1551.16	3179.62	3101.19
Audiobeam	78.16	168.11	180.76
ChannelVocoder	881.73	1892.12	1971.79
Filterbank	434.96	909.75	1039.9
TargetDetect	30.6	78.13	70.39
FMRadio	153.22	338.67	390.07
FFT (coarse)	8.24	16.56	17.72
FFT (medium)	57.72	118.65	106.73
RateConvert	112.86	372.7	221
TDE	10.46	21.03	38.18
MatrixMult (fine)	48.91	109.27	133.5
MatrixMult (coarse)	85.61	134.1	214.81

Table 7.6: Throughput values for dynamic load-balancing under load

7.4.2 Adapting to Changes in Core Availability

To determine how well dynamic load-balancing responds to changes in core availability, benchmarks are initially run across all cores. A process is then executed on one of the cores that steals processing time. Ideally, dynamic load-balancing would shift work away from the core onto other cores in order to minimize the performance impact of a competing process. Throughput is measured once dynamic load-balancing has had some time to settle. For comparative purposes, throughput is also measured with dynamic load-balancing turned off when the competing process is executing. If dynamic load-balancing works as intended, then the throughput with dynamic load-balancing turned on should be higher than the throughput with dynamic load-balancing turned off.

Throughput results are shown in Table 7.6. Unfortunately, dynamic load-balancing does not appear to work as intended. In fact, the throughput with dynamic load-balancing turned on is often worse than the throughput with dynamic load-balancing turned off, which is a very surprising result.

Close examination reveals that dynamic load-balancing interacts with the Linux scheduler in unintended ways. When a thread shares a core with another process, dynamic load-balancing assumes that the steady-state execution times for the thread will become uniformly longer. Unfortunately, this isn't the case since the Linux scheduler makes use of time slices to multiplex processes and threads. When a thread is

given a time slice, the thread has unrestricted access to its core, allowing steady-state execution to proceed at full-speed. When there is a context switch to a competing process, the thread sees this as a single steady-state that takes an extraordinarily long time to execute. Thus, instead of seeing steady-state execution times that are uniformly longer, the thread sees a stream of ordinary steady-state execution times that is periodically punctuated by individual steady-state execution times that are extraordinarily long.

This has negative effects on dynamic load-balancing. When dynamic load-balancing sees an extraordinarily long steady-state execution time, it moves large amounts of stateless work away from the core. When steady-state execution times immediately return to normal, dynamic load-balancing moves the stateless work back onto the core. Because context switches happen periodically, extraordinarily long steady-state execution times also happen periodically, causing this process to be repeated indefinitely. Large amounts of stateless work is therefore sloshed back and forth between the cores, which impairs overall performance.

Essentially, dynamic load-balancing attempts to discover whether multiple processes are being multiplexed on the same core, while the Linux scheduler makes it difficult for the processes themselves to detect that this might be happening. This nullifies the effectiveness of dynamic load-balancing. Dynamic load-balancing can't effectively detect when cores are being shared, and as such dynamic load-balancing is unable to effectively compensate when it happens.

Assuming that dynamic load-balancing did not suffer from this effect, it is unknown whether dynamic load-balancing would be able to fully mitigate the performance impact of losing a core. Note that even if dynamic load-balancing moves all stateless filter work away from a core, the core still needs to execute the steady-state barrier. This core has the potential to become a bottleneck if much of the core's processing time is spent on competing processes. However, even if dynamic load-balancing could not fully mitigate the performance impact of losing a core, it has the potential for substantial performance improvement over not having dynamic load-balancing.

Benchmark	Stateless filter instances	Load-Balance Clock Cycles
Radar (fine)	8	30196
Audiobeam	15	53224
ChannelVocoder	34	8497
Filterbank	24	57988
TargetDetect	4	30452
FMRadio	3	14842
FFT (coarse)	1	5615
FFT (medium)	1	6129
RateConvert	1	22700
TDE	1	16730
MatrixMult (fine)	1	9967
MatrixMult (coarse)	2	10847

Table 7.7: Clock cycles per load-balancing pass

7.4.3 Run-Time Overhead

Dynamic load-balancing has two sources of run-time overhead: run-time profiling and load-balancing passes.

To generate run-time profiles, each core times how long it takes to execute its iterations of each stateless filter. Timing is done using the Time Stamp Counter, which takes roughly 60 cycles to read once. Run-time profiling therefore adds an overhead of roughly 120 cycles to each stateless filter on each core. This overhead is usually pretty insignificant. A large number of stateless filters would be needed to make this overhead substantial.

Load-balancing passes are also fairly inexpensive as the load-balancing algorithm is constructed to be very simple. The number of cycles needed for a load-balancing pass is roughly correlated with the number of stateless filters in the stream program. Table 7.7 lists the number of stateless filters in each benchmark and the average number of cycles a load-balancing pass takes for each benchmark.

Because run-time profiling and load-balancing passes are executed periodically and not on every steady-state iteration, the overhead introduced is amortized over multiple steady-state iterations. This further reduces the run-time overhead of dynamic load-balancing.

Figure 7-10 breaks down execution time into time spent performing useful work, time spent in the barrier, and time spent in a load-balancing pass. The time spent

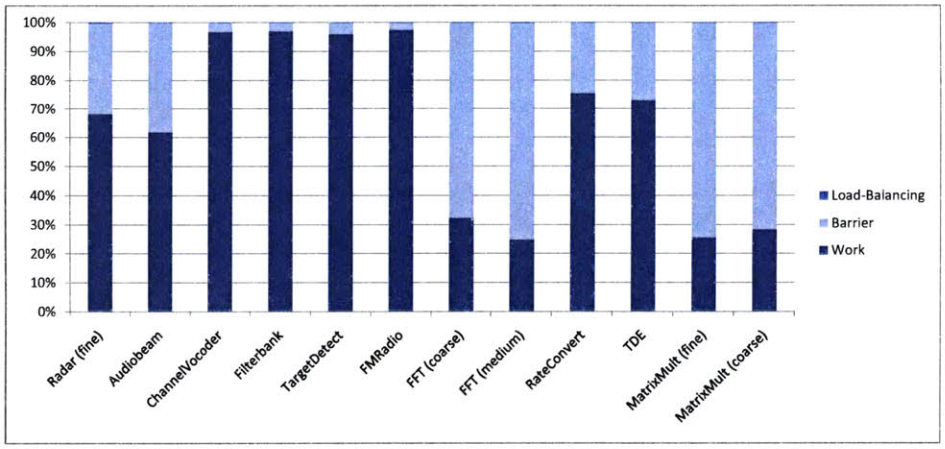


Figure 7-10: Breakdown of execution costs with dynamic load-balancing

in a load-balancing pass is so insignificant that it is largely invisible in the figure. Note that this suggests that run-time profiling and load-balancing passes can be executed far more often than the default settings presently allow. Run-time profiling should be executable on every steady-state iteration. Load-balancing passes may also be executable on every steady-state iteration, but this may be undesirable as multiple run-time profiles are necessary for accuracy in run-time measurements. Running load-balancing passes too often will force load-balancing to depend on run-time measurements that may not be accurate.

7.5 Summary

Optimized filter fission was found to have a significant impact on the benchmarks for which it is applicable. For these benchmarks, optimized filter fission improved parallel speedup to approximately the ideal of 16x. Optimized filter fission improved parallel speedups by reducing the communication overhead associated with filter fission. Optimized filter fission also improved parallel speedups by increasing the amount of steady-state work in each benchmark, which help to better amortize the cost of the steady-state barrier.

Dynamic load-balancing was found to be very successful in fixing inaccurate static

load-balancing. Dynamic load-balancing was able to improve performance for nearly all of the tested benchmarks. Most of the benchmarks required only minor adjustments to the static schedule, while a couple of the benchmarks required large-scale adjustments. This demonstrates that dynamic load-balancing is capable of performing both fine-grained and large-scale adjustments to the static schedule. Load-balancing converged very quickly, requiring only a small number of load-balancing passes to reach a load-balanced state.

Unfortunately, dynamic load-balancing was not as successful in compensating for cores that are shared with other processes. Ideally, dynamic load-balancing would shift work away from these cores. However, unintended interactions with the Linux scheduler prevents dynamic load-balancing from detecting when cores are shared with other processes. Dynamic load-balancing is therefore largely unable to compensate for these cores. In fact, performance is sometimes degraded as the unintended interactions with the Linux scheduler causes dynamic load-balancing to repeatedly move large amounts of work back and forth between cores.

Dynamic load-balancing was found to have very little run-time overhead. The overhead introduced by dynamic load-balancing is so insignificant that dynamic load-balancing can be set to occur much more frequently than the default settings allow.

Chapter 8

Related Work

Numerous approaches have been developed to hybridize static and dynamic scheduling. In [6] and [19], two classes of hybrid scheduling are identified. *Static allocation* is a hybrid scheduling approach where actors are assigned to cores at compile-time, but cores determine the order of actor executions at run-time. Many dataflow machines operate in this fashion, such as the Monsoon architecture [24]. *Self-timed* is another hybrid scheduling approach where the compiler determines both core assignments and execution ordering for actors, but does not determine the exact timing of actor executions. At run-time, each core waits for data to be available for the next actor in its ordered list before executing the actor. This scheduling approach is commonly used when there is no hardware support for scheduling beyond synchronization primitives. This approach is utilized in the Gabriel architecture [4].

These hybrid scheduling approaches differ significantly from the approach taken in this thesis in that they do not allow actors to be moved between cores. Scheduling decisions made at run-time allow for a limited amount of load-balancing. However, if actors are assigned to cores such that cores are severely imbalanced, these scheduling approaches do not provide any mechanisms for redistributing actors.

Flexstream [15] is a recent hybrid scheduling approach that does allow actors to be moved between cores. Flexstream partitions the stream graph at compile-time, generating roughly load-balanced partitions that are assigned to cores. At run-time, whenever cores become unavailable, Flexsteram redistributes the partitions into fewer

partitions. This moves actors to redistribute load across the remaining cores.

Flexstream has a number of similarities with the scheduling approach taken in this thesis. A static schedule is generated that is adjusted at run-time whenever cores become unavailable. Adjustments are inexpensive and are only performed whenever needed, resulting in very little run-time overhead. The use of static scheduling allows for sophisticated optimizations, such as globally minimizing communication and buffering requirements. Flexstream also has a number of significant differences. Flexstream makes no attempt to fix load imbalances that might exist after static load-balancing. Flexstream partitions may be significantly imbalanced if static work estimates are inaccurate. Partitions also can't be adjusted in a fine-grained manner as entire actors must be moved between partitions. Partitions in Flexstream therefore tend to be slightly imbalanced.

An alternative StreamIt dynamic load-balancing system is described in [9]. This system approaches filter scheduling as a resource allocation problem. Each filter is assigned a resource budget that determines its priority in execution. Filters with larger resource budgets are given higher priority in execution. If a filter runs out of input, it can give part of its resource budget to source filters to give them higher priority. If a filter runs out of space in its output channel, it can give part of its resource budget to destination filters to give them higher priority. Filters therefore make decentralized decisions on how resource budgets are allocated, which performs a rough form of load-balancing. Note that this system exists only in simulation and has not been implemented.

Previous work in StreamIt minimized the communication overhead associated with filter fission by applying an optimization called *judicious fission* [11]. Judicious fission avoids fissing filters across all cores as this introduces a significant amount of communication between cores. In each level of the stream graph, judicious fission looks for task-parallel stateless filters. Task-parallel stateless filters are fissed such that the cores are divided proportionally amongst the stateless filters. The proportion of cores that each stateless filter receives depends on static work estimates for the filters. Stateless filters with more work are fissed across more cores. This divides

the work of the task-parallel stateless filters such that each core receives roughly the same amount of work. This process is repeated for all levels of the stream graph.

By avoiding the fission of filters across all cores, judicious fission was found to be very successful at reducing the communication overhead associated with filter fission. Unfortunately, judicious fission was not explored during the course of this thesis as it complicates dynamic load-balancing. For a filter that is fissed on only a subset of the cores, dynamic load-balancing would only be able to move filter iterations between these subset of cores. This places constraints on dynamic load-balancing. Taking these constraints into account would require a more sophisticated load-balancing algorithm, which would likely have a higher run-time cost.

Rather than performing judicious fission, the SMP compiler backend described in this thesis opts to fiss stateless filters across all cores. As described, this adds significant communication overhead, though the optimizations discussed in Chapter 6 help to reduce the amount of communication overhead. One advantage of fissing stateless filters across all cores is that cores are guaranteed to receive the same amount of work from each stateless filter. Though judicious fission attempts to give cores the same amount of work from each stream graph level, this isn't always guaranteed. For example, inaccurate static work estimates can cause judicious fission to give cores unequal amounts of work from each level. Fissing stateless filters across all cores can therefore result in better static load-balancing than judicious fission.

Previous work in optimizing the execution of stream programs on commodity processors is discussed in [13]. The authors detail how stream programs can take advantage of the unique architectural features of commodity processors, such as shared memory, multi-level caching, and instruction-level stream enhancements. Some of the discussed techniques are utilized in this thesis. The authors place heavy emphasis on efficient scatter and gather operations, non-temporal loads and stores to avoid interferance with caches, and parallel execution of computation and communication.

Chapter 9

Conclusion

Static scheduling and dynamic scheduling each have their own advantages and disadvantages. Static scheduling with dynamic adjustments attempts to combine the best of both types of scheduling by starting with a static schedule and dynamically adjusting it to run-time conditions. Dynamic adjustments allow load imbalances in the static schedule to be corrected at run-time. Dynamic adjustments also attempt to compensate for cores that become unavailable or shared with other processes. This scheduling approach is unique in that it is mostly static with inexpensive dynamic adjustments, which results in very low run-time overhead.

This approach was shown to be mostly successful in this thesis. Dynamic load-balancing was very effective at correcting load imbalances in the static schedule, which in some cases resulted in large improvements in run-time performance. Dynamic load-balancing was unable to compensate for cores that become unavailable or shared with other processes. However, this was found to be a flaw in the dynamic load-balancing algorithm used in this thesis rather than a flaw in the overall approach of static scheduling with dynamic adjustments. Due to unintended interactions with the Linux scheduler, dynamic load-balancing was unable to detect when a core is unavailable or shared, which prevented dynamic load-balancing from compensating. If the OS can provide detailed information on how cores are utilized, future work may be able to implement an improved dynamic load-balancing algorithm that can effectively detect and compensate for cores that are unavailable or shared.

This thesis contributes methods for adjusting a static schedule at run-time, a capability that may be useful in a number of other contexts. This thesis also contributes important optimizations on filter fission that reduce the communication overhead introduced by filter fission. These optimizations appear to have a dramatic impact on the parallel performance of StreamIt programs. These contributions are likely to be exploited in future StreamIt work.

Bibliography

[1] Sitij Agrawal, William Thies, and Saman Amarasinghe. Optimizing stream programs using linear state space analysis. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 126–136, New York, NY, USA, 2005. ACM Press.

[2] Abdulbasier Aziz. Image-based motion estimation in a stream programming language. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, Jun 2007.

[3] Shuvra S. Bhattacharyya and Edward A. Lee. Scheduling synchronous dataflow graphs for efficient looping. *J. VLSI Signal Process. Syst.*, 6(3):271–288, 1993.

[4] J.C. Bier, E.E. Goei, W.H. Ho, P.D. Lapsley, M.P. O'Reilly, G.C. Sih, and E.A. Lee. Gabriel: A design environment for dsp. *IEEE Micro*, 10(5):28–45, 1990.

[5] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[6] Joseph T Buck. Scheduling dynamic dataflow graphs with bounded memory. Technical report, Berkeley, CA, USA, 1993.

[7] Jiawen Chen, Michael I. Gordon, William Thies, Matthias Zwicker, Kari Pulli, and Frédo Durand. A reconfigurable architecture for load-balanced rendering. In *SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 2005.

[8] M. Drake, H. Hoffmann, R. Rabbah, and S. Amarasinghe. Mpeg-2 decoding in a stream programming language. pages 10 pp.+, 2006.

[9] Eric Todd Fellheimer. Dynamic load-balancing of streamit cluster computations by. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2006.

[10] Michael Gordon. A stream-aware compiler for communication-exposed architectures. S.m. thesis, Massachusetts Institute of Technology, Cambridge, MA, Aug 2002.

[11] Michael Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct 2006.

[12] R. Govindarajan, G.R. Gao, and P. Desai. Minimizing memory requirements in rate-optimal schedules, 1994.

[13] Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–354, Washington, DC, USA, 2005. IEEE Computer Society.

[14] Soonhoi Ha and Edward A. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers*, 46(7):768–778, 1997.

[15] Amir Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Parallel Architectures and Compilation Techniques*, 2009.

[16] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003.

[17] Michal Karczmarek. Constrained and phased scheduling of synchronous data flow graphs for streamit language. S.m. thesis, Massachusetts Institute of Technology, Cambridge, MA, Dec 2002.

[18] Andrew A. Lamb, William Thies, and Saman Amarasinghe. Linear analysis and optimization of stream programs. In *In PLDI*, 2003.

[19] Edward Ashford Lee. Scheduling strategies for multiprocessor real-time dsp. pages 1279–1283, 1989.

[20] Edward Ashford Lee and David. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36:24–35, 1987.

[21] William R. Mark, Steven R. Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM Press.

[22] John M. Mellor-crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9:21–65, 1991.

[23] Hyunok Oh, Nikil Dutt, and Soonhoi Ha. Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In *ASP-DAC '06: Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 497–502, Piscataway, NJ, USA, 2006. IEEE Press.

[24] Gregory M. Papadopoulos. *Implementation of a general-purpose dataflow multiprocessor.* MIT Press, Cambridge, MA, USA, 1991.

[25] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 115–126, New York, NY, USA, 2005. ACM Press.

[26] William Thies. *Language and Compiler Support for Stream Programs.* Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, Feb 2009.

[27] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr 2002.

[28] William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe. Teleport messaging for distributed stream programs. In *Symposium on Principles and Practice of Parallel Programming*, Chicago, Illinois, Jun 2005.

[29] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 200–209, Washington, DC, USA, 2009. IEEE Computer Society.

[30] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Synergistic execution of stream programs on multicores with accelerators. *SIGPLAN Not.*, 44(7):99–108, 2009.

[31] Xin David Zhang. A streaming computation framework for the cell processor. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, Aug 2007.